MATHEMATICAL FOUNDATIONS
FOR
DATA ANALYSIS

JEFF M. PHILLIPS

2018

copyright: Jeff M. Phillips

Preface

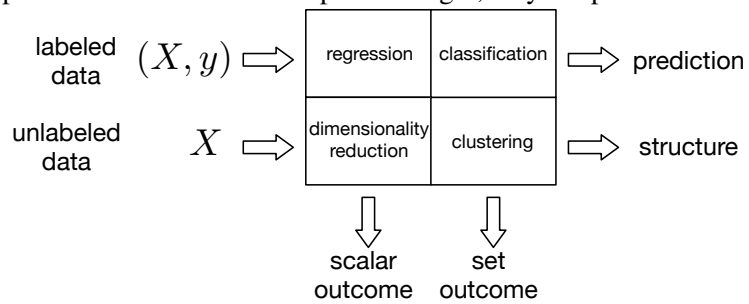
This book is meant for use with a self-contained course that introduces many basic principles and techniques needed for modern data analysis. In particular, this course was designed as preparation for students planning to take rigorous Machine Learning and Data Mining courses. With this goal in mind, it both introduces key conceptual tools which are for most students are ultimately helpful to see multiple times. It also introduces the generic versions of the most basic techniques that comprise the backbone of modern data analysis.

Notation. Consistent, clear, and crisp mathematical notation is essential for intuitive learning. The domains which comprise modern data analysis (e.g., statistics, machine learning, algorithms) have matured somewhat separately with their own conventions for ways to write the same or similar concepts. Moreover, it is commonplace for researchers to adapt notation to best highlight ideas within specific papers. As such, much of the existing literature on the topics covered in this book is all over the place, inconsistent, and as a whole confusing. This text attempts to establish a common, simple, and consistent notation for these ideas, yet to still not veer too far from how concepts are consistently represented in research literature, and as they will be in more advanced courses. Indeed the most prevalent sources of confusion in earlier uses of this text in class have arisen around overloaded notation.

Interaction with other courses. It is recommended that students taking this class have calculus and a familiarity with programming and algorithms. They should have also taken some probability and/or linear algebra; but we also review key concepts in these areas, so as to keep the book more self-contained. Thus, it may be appropriate for students to take these classes simultaneously. If appropriately planned for, it is the hope that this course could be taken at the sophomore level so that more rigorous and advanced data analysis classes can already be taken during the junior year.

Although we touch on Bayesian Inference, we do not cover most of classical statistics; neither frequentist hypothesis testing or the similar Bayesian perspectives. Most universities have well-developed courses on these topics which while also very useful, provide a complimentary view of data analysis. Classical statistical modeling approaches are often essential when a practitioner needs to provide some modeling assumptions to harness maximum power from limited data. But in the era of big data this is not always necessary. Rather, the topics in this course provide tools for using some of the data to help choose the model.

Scope and topics. Vital concepts introduced include concentration of measure and PAC bounds, cross-validation, gradient descent, and principal component analysis. These ideas are essential for modern data analysis, but not often taught in other introductory mathematics classes in a computer science or math department. Or if these concepts are taught, they are presented in a very different context.



We also survey basic techniques in supervised (regression and classification) and unsupervised (principal component analysis and clustering) learning. We make an effort to keep the presentation and concepts on these topics simple. We mainly stick to those which attempt to minimize sum of squared errors, and do not go much into regularization. We stick to classic but magical algorithms like Lloyd's algorithm for k-means,

copyright: Jeff M. Phillips

the power method for eigenvectors, and perceptron for linear classification. For many students (especially those in a computer science program), these are the first iterative, non-discrete algorithms they will have encountered.

On data. While this course is mainly focused on a mathematical preparation, what would data analysis be without data? As such we provide discussion on how to use these tools and techniques on actual data, with examples given in python. We choose python since it has increasingly many powerful libraries often with efficient backends in C. So for most data sets, this provides the proper interface for working with these tools.

But arguably more important than writing the code itself is a discussion on when and when-not to use techniques from the immense toolbox available. This is one of the main ongoing questions a data scientist must ask. And so, the text attempts to introduce the readers to this ongoing discussion.

Jeff M. Phillips
Salt Lake City, August 2018

 copyright: Jeff M. Phillips

Contents

1 Probability Review **7**
- 1.1 Sample Spaces . . . 7
- 1.2 Conditional Probability and Independence . . . 8
- 1.3 Density Functions . . . 9
- 1.4 Expected Value . . . 10
- 1.5 Variance . . . 11
- 1.6 Joint, Marginal, and Conditional Distributions . . . 12

2 Bayes' Rule **17**
- 2.1 Bayes' Rule . . . 17
  - 2.1.1 Model Given Data . . . 18
- 2.2 Bayesian Inference . . . 19

3 Convergence **23**
- 3.1 Sampling and Estimation . . . 23
- 3.2 Probably Approximately Correct (PAC) . . . 25
- 3.3 Concentration of Measure . . . 26

4 Linear Algebra Review **33**
- 4.1 Vectors and Matrices . . . 33
- 4.2 Addition . . . 34
- 4.3 Multiplication . . . 34
- 4.4 Norms . . . 36
- 4.5 Linear Independence . . . 37
- 4.6 Rank . . . 38
- 4.7 Inverse . . . 38
- 4.8 Orthogonality . . . 38

5 Linear Regression **43**
- 5.1 Simple Linear Regression . . . 43
- 5.2 Linear Regression with Multiple Explanatory Variables . . . 45
- 5.3 Polynomial Regression . . . 49
- 5.4 Cross Validation . . . 50

6 Gradient Descent **57**
- 6.1 Functions . . . 57
- 6.2 Gradients . . . 58
- 6.3 Gradient Descent . . . 59
  - 6.3.1 Learning Rate . . . 60
- 6.4 Fitting a Model to Data . . . 63
  - 6.4.1 Least Mean Squares Updates for Regression . . . 64
  - 6.4.2 Decomposable Functions . . . 65

7 Principal Component Analysis **69**
7.1 Data Matrices . . . 69
7.1.1 Projections . . . 70
7.1.2 SSE Goal . . . 71
7.2 Singular Value Decomposition . . . 71
7.2.1 Best Rank-k Approximation . . . 74
7.3 Eigenvalues and Eigenvectors . . . 75
7.4 The Power Method . . . 77
7.5 Principal Component Analysis . . . 79
7.6 Multidimensional Scaling . . . 79

8 Clustering **83**
8.1 Voronoi Diagrams . . . 83
8.1.1 Delaunay Triangulation . . . 84
8.1.2 Connection to Clustering . . . 86
8.2 k-Means Clustering . . . 86
8.3 Lloyd's Algorithm . . . 86
8.3.1 Extensions to the k-Means Problem and Lloyd's Algorithm . . . 88
8.4 Mixture of Gaussians . . . 88
8.4.1 Expectation-Maximization . . . 89
8.5 Mean Shift Clustering . . . 90

9 Classification **93**
9.1 Linear Classifiers . . . 93
9.1.1 Loss Functions . . . 95
9.1.2 Cross Validation and Regularization . . . 96
9.2 Perceptron Algorithm . . . 97
9.3 Kernels . . . 100
9.3.1 The Dual: Mistake Counter . . . 101
9.3.2 Feature Expansion . . . 101
9.3.3 Support Vector Machines . . . 102
9.4 kNN Classifiers . . . 103
9.5 Neural Networks . . . 103

1 Probability Review

Probability is a critical tool for modern data analysis. It arises in dealing with uncertainty, in randomized algorithms, and in Bayesian analysis. To understand any of these concepts correctly, it is paramount to have a solid and rigorous statistical foundation. Here we review some key definitions.

1.1 Sample Spaces

We define probability through set theory, starting with a *sample space* Ω. This represents the set of all things that might happen in the setting we consider. One such potential outcome $\omega \in \Omega$ is a *sample outcome*, it is an element of the set Ω. We are usually interested in an *event* that is a subset $A \subseteq \Omega$ of the sample space.

Example: Discrete Sample Space for a 6-Ssided Die

Consider rolling a single fair, 6-sided die. Then $\Omega = \{1, 2, 3, 4, 5, 6\}$. One roll may produce an outcome $\omega = 3$, rolling a 3. An event might be $A = \{1, 3, 5\}$, any odd number.
The probability of rolling an odd number is then $\textbf{Pr}(A) = |\{1, 3, 5\}|/|\{1, 2, 3, 4, 5, 6\}| = 1/2$.

A *random variable* $X : \Omega \to S$ is a **function** which maps from the sample space Ω to a domain S. In many cases $S \subseteq \mathbb{R}$, where $\mathbb{R}$ is the space of real numbers.

Example: Random Variables for a Fair Coin

Consider flipping a fair coin with $\Omega = \{H, T\}$. If I get a head H, then I get 1 point, and if I get a T, then I get 4 points. This describes the random variable X, defined $X(H) = 1$ and $X(T) = 4$.

The *probability* of an event $\textbf{Pr}(A)$ satisfies the following properties:

- $0 \leq \textbf{Pr}(A) \leq 1$ for any A,
- $\textbf{Pr}(\Omega) = 1$, and
- The probability of the union of disjoint events is equivalent to the sum of their individual probabilities. Formally, for any sequence $A_1, A_2, \ldots$ where for all $i \neq j$ that $A_i \cap A_j = \emptyset$, then

$$\textbf{Pr}\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \textbf{Pr}(A_i).$$

Example: Probability for a Biased Coin

Now consider flipping a biased coin with two possible events $\Omega = \{H, T\}$ (i.e., heads $H = A_1$ and tails $T = A_2$). The coin is biased so the probabilities $\textbf{Pr}(H) = 0.6$ and $\textbf{Pr}(T) = 0.4$ of these events are not equal. However we notice still $0 \leq \textbf{Pr}(T), \textbf{Pr}(H) \leq 1$, and that $\textbf{Pr}(\Omega) = \textbf{Pr}(H \cup T) = \textbf{Pr}(H) + \textbf{Pr}(T) = 0.6 + 0.4 = 1$. That is the sample space Ω is union of these two events, which cannot both occur (i.e., $H \cap T = \emptyset$) so they are disjoint. Thus Ω's probability can be written as the sum of the probability of those two events.

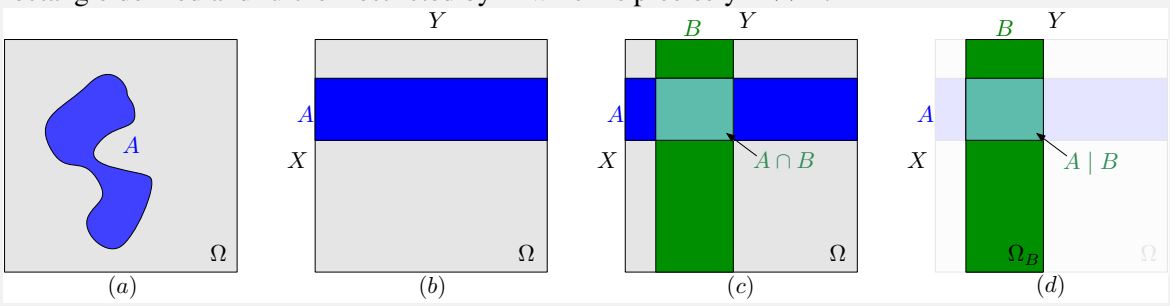
Sample spaces Ω can also be continuous, representing some quantity like water, time, or land mass which does not have discrete quantities. All of the above definitions hold for this setting.

Example: Continuous Sample Space

Assume you are riding a Swiss train that is always on time, but its departure is only specified to the minute (specifically, 1:37 pm). The true departure is then in the state space $\Omega = [1{:}37{:}00, 1{:}38{:}00)$. A continuous event may be $A = [1{:}37{:}00 - 1{:}37{:}40)$, the first 40 seconds of that minute.
Perhaps the train operators are risk averse, so $\mathbf{Pr}(A) = 0.80$. That indicates that 0.8 fraction of trains depart in the first $2/3$ of that minute (less than the 0.666 expected from a uniform distribution).

Geometry of Sample Spaces

It may be useful to generically imagine a sample space Ω as a square. Then, as shown in (a), an event A may be an arbitrary subset of this space.
When the sample space describes the joint sample space over two random variables X and Y, then it may be convenient to parameterize Ω so that the X value is represented along one side of the square, and the Y value along the other, as in (b). Then for an event $A \subset X$ which only pertains to the random variable X is represented as a rectangular strip defined by A's intersection with the domain of X.
If there is also an event $B \subset Y$ that only pertains to random variable Y, then another rectangular strip in the other direction defined by B's intersection with the domain of Y can be drawn as in (c). When these events are independent, then these strips intersect only in a another rectangle $A \cap B$. When X and Y are independent, then *all such strips*, defined by events $A \subset X$ and $B \subset Y$ intersect in a rectangle. If the events are not independent, then the associated picture will not look as clear, like in (a).
Given such independent events $A \subset X$ and $B \subset Y$, it is easy to see that $A \mid B$ can be realized, as in (d), with the rectangle $A \cap B$ restricted to the strip defined by B. Furthermore, imagining the area as being proportional to probability, it is also easy to see that $\mathbf{Pr}(A \mid B) = \mathbf{Pr}(A \cap B)/\mathbf{Pr}(B)$ since the strip B induces a new restricted sample space Ω_B, and an event only occurs in the strip-induced rectangle defined and further-restricted by A which is precisely $A \cap B$.



1.2 Conditional Probability and Independence

Now consider two events A and B. The *conditional probability* of A given B is written $\mathbf{Pr}(A \mid B)$, and can be interpreted as the probability of A, restricted to the setting where we know B is true. It is defined in simpler terms as $\mathbf{Pr}(A \mid B) = \frac{\mathbf{Pr}(A \cap B)}{\mathbf{Pr}(B)}$, that is the probability A and B are both true, divided by (normalized by) the probability B is true.

Two **events** A and B are *independent* of each other if and only if

$$\mathbf{Pr}(A \mid B) = \mathbf{Pr}(A).$$

copyright: Jeff M. Phillips

Equivalently they are independent if and only if $\textbf{Pr}(B \mid A) = \textbf{Pr}(B)$ or $\textbf{Pr}(A \cap B) = \textbf{Pr}(A)\textbf{Pr}(B)$. By algebraic manipulation, it is not hard to see these are all equivalent properties. This implies that knowledge about B has no effect on the probability of A (and vice versa from A to B).

Example: Conditional Probability

Consider the two random variables. T is 1 if a test for cancer is positive, and 0 otherwise. Variable C is 1 if a patient has cancer, and 0 otherwise. The joint probability of the events is captured in the following table:

		cancer	no cancer
		$C = 1$	$C = 0$
tests positive for cancer	$T = 1$	0.1	0.02
tests negative for cancer	$T = 0$	0.05	0.83

Note that the sum of all cells (the joint sample space Ω) is 1. The conditional probability of having cancer, given a positive test is $\textbf{Pr}(C = 1 \mid T = 1) = \frac{0.1}{0.1+0.02} = 0.8333$. The probability of cancer (ignoring the test) is $\textbf{Pr}(C = 1) = 0.1 + 0.05 = 0.15$. Since $\textbf{Pr}(C = 1 \mid T = 1) \neq \textbf{Pr}(C = 1)$, then events $T = 1$ and $C = 1$ are not independent.

Two **random variables** X and Y are *independent* if and only if, for *all* possible events $A \subseteq \Omega_X$ and $B \subseteq \Omega_Y$ that A and B are independent: $\textbf{Pr}(A \cap B) = \textbf{Pr}(A)\textbf{Pr}(B)$.

1.3 Density Functions

Discrete random variables can often be defined through tables (as in the above cancer example). Or we can define a function $f_X(k)$ as the probability that random variable X is equal to k. For continuous random variables we need to be more careful: we will use calculus. We will next develop probability density functions (pdfs) and cumulative density functions (cdfs) for continuous random variables; the same constructions are sometimes useful for discrete random variables as well, which basically just replace a integral with a sum.

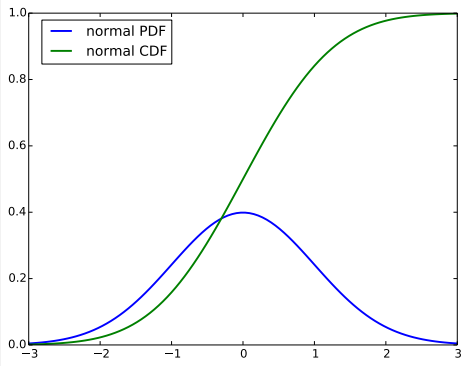
We consider a continuous sample space Ω, and a random variable X defined on that sample space. The probability density function of a random variable X is written f_X. It is defined with respect to any event A so that $\textbf{Pr}(X \in A) = \int_{\omega \in A} f_X(\omega) \mathrm{d}\omega$. The value $f_X(\omega)$ *is not equal to* $\textbf{Pr}(X = \omega)$ in general, since for continuous functions $\textbf{Pr}(X = \omega) = 0$ for any single value $\omega \in \Omega$. Yet, we can interpret f_X as a *likelihood* function; its value has no units, but they can be compared and larger ones are more likely.

Next we will define the *cumulative density function* $F_X(t)$; it is the probability that X takes on a value of t or smaller. Here it is typical to have $\Omega = \mathbb{R}$, the set of real numbers. Now define $F_X(t) = \int_{\omega=-\infty}^{t} f_X(\omega) \mathrm{d}\omega$.

We can also define a pdf in terms of a cdf as $f_X(\omega) = \frac{\mathrm{d}F_X(\omega)}{\mathrm{d}\omega}$.

Example: Normal Random Variable

A *normal* random variable X is a very common distribution to model noise. It has domain $\Omega = \mathbb{R}$. Its pdf is defined $f_X(\omega) = \frac{1}{\sqrt{2\pi}} \exp(-\omega^2/2) = \frac{1}{\sqrt{2\pi}} e^{-\omega^2/2}$, and its cdf has no closed form solution. We have plotted the cdf and pdf in the range $[-3, 3]$ where most of the mass lies:




```
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt
from scipy.stats import norm
import numpy as np
import math

mu = 0
variance = 1
sigma = math.sqrt(variance)
x = np.linspace(-3, 3, 201)

plt.plot(x, norm.pdf((x-mu)/sigma),linewidth=2.0, label='normal␣PDF')
plt.plot(x, norm.cdf((x-mu)/sigma),linewidth=2.0, label='normal␣CDF')
plt.legend(bbox_to_anchor=(.35,1))

plt.savefig('Gaussian.pdf', bbox_inches='tight')
```

1.4 Expected Value

The expected value of a random variable X in a domain Ω is a very important constant, basically a weighted average of Ω, weighted by the range of X. For a discrete random variable X it is defined as the sum over all outcomes ω in the sample space, or their value times their probability

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} (\omega \cdot \mathbf{Pr}[X = \omega]).$$

For a continuous random variable X it is defined

$$\mathbf{E}[X] = \int_{\omega \in \Omega} \omega \cdot f_X(\omega) \mathrm{d}\omega.$$

Linearity of Expectation: An important property of expectation is that it is a linear operation. That means for two random variables X and Y we have $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$. For a scalar value α, we also $\mathbf{E}[\alpha X] = \alpha \mathbf{E}[X]$.

copyright: Jeff M. Phillips

Example: Expectation

A fair die has a sample space of $\Omega = \{\omega_1 = 1, \omega_2 = 2, \omega_3 = 3, \omega_4 = 4, \omega_5 = 5, \omega_6 = 6\}$, and the probability of each outcome w_i is $\mathbf{Pr}[w_i] = 1/6$. The expected value able of a random variable D of the result of a roll of such a die is

$$\mathbf{E}[D] = \sum_{\omega_i \in \Omega} \omega_i \cdot \mathbf{Pr}[D = \omega_i] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = \frac{21}{6} = 3.5$$

Example: Linearity of Expectation

Let H be the random variable of the height of a man in meters without shoes. Let the pdf f_H of H be a normal distribution with expected value $\mu = 1.755$m and with standard deviation 0.1m. Let S be the random variable of the height added by wearing a pair of shoes in centimeters (1 meter is 100 centimeters), its pdf is given by the following table:

$S = 1$	$S = 2$	$S = 3$	$S = 4$
0.1	0.1	0.5	0.3

Then the expected height of someone wearing shoes in centimeters is

$$\mathbf{E}[100 \cdot H + S] = 100 \cdot \mathbf{E}[H] + \mathbf{E}[S] = 100 \cdot 1.755 + (0.1 \cdot 1 + 0.1 \cdot 2 + 0.5 \cdot 3 + 0.3 \cdot 4) = 175.5 + 3 = 178.5$$

Note how the linearity of expectation allowed us to decompose the expression $100 \cdot H + S$ into its components, and take the expectation of each one individually. This trick is immensely powerful when analyzing complex scenarios with many factors.

1.5 Variance

The *variance* of a random variable X describes how spread out it is, with respect to its mean $\mathbf{E}[X]$. It is defined

$$\begin{aligned} \mathbf{Var}[X] &= \mathbf{E}[(X - \mathbf{E}[X])^2] \\ &= \mathbf{E}[X^2] - \mathbf{E}[X]^2. \end{aligned}$$

The equivalence of those two above common forms above uses that $\mathbf{E}[X]$ is a fixed scalar:

$$\mathbf{E}[(X - \mathbf{E}[X])^2] = \mathbf{E}[X^2 - 2X\mathbf{E}[X] + \mathbf{E}[X]^2] = \mathbf{E}[X^2] - 2\mathbf{E}[X]\mathbf{E}[X] + \mathbf{E}[X]^2 = \mathbf{E}[X^2] - \mathbf{E}[X]^2.$$

For any scalar $\alpha \in \mathbb{R}$, then $\mathbf{Var}[\alpha X] = \alpha^2 \mathbf{Var}[X]$.

Note that the variance does not have the same units as the random variable or the expectation, it is that unit squared. As such, we also often discuss the *standard deviation* $\sigma_X = \sqrt{\mathbf{Var}[X]}$. A low value of $\mathbf{Var}[X]$ or σ_X indicates that most values are close to the mean, while a large value indicates that the data has a higher probability of being further from the mean.

 copyright: Jeff M. Phillips

Example: Variance

Consider again the random variable S for height added by a shoe:

$S=1$	$S=2$	$S=3$	$S=4$
0.1	0.1	0.5	0.3

Its expected value is $\mathbf{E}[S] = 3$ (a fixed scalar), and its variance is

$$\begin{aligned}\mathbf{Var}[S] &= 0.1 \cdot (1-3)^2 + 0.1 \cdot (2-3)^2 + 0.5 \cdot (3-3)^2 + 0.3 \cdot (4-3)^2 \\ &= 0.1 \cdot (-2)^2 + 0.1 \cdot (-1)^2 + 0 + 0.3(1)^2 = 0.4 + 0.1 + 0.3 = 0.8.\end{aligned}$$

Then the standard deviation is $\sigma_S = \sqrt{0.8} \approx 0.894$.

The *covariance* of two random variables X and Y is defined $\mathbf{Cov}[X,Y] = \mathbf{E}[(X - \mathbf{E}[X])(Y - \mathbf{E}[Y])]$. It measures how much these random variables vary in accordance with each other; that is, if both are consistently away from the mean at the same time (in the same direction), then the covariance is high.

1.6 Joint, Marginal, and Conditional Distributions

We now extend some of these concepts to more than one random variable. Consider two random variables X and Y. Their *joint pdf* is defined $f_{X,Y} : \Omega_X \times \Omega_Y \to [0, \infty]$ where for discrete random variables this is defined by the probability $f_{X,Y}(x,y) = \mathbf{Pr}(X = x, Y = y)$. In this discrete case, the domain of $f_{X,Y}$ is restricted so $f_{X,Y} \in [0,1]$ and so $\sum_{x,y \in X \times Y} f_{X,Y}(x,y) = 1$, e.g., the sum of probabilities over the joint sample space is 1.

Similarly, when $\Omega_X = \Omega_Y = \mathbb{R}$, the *joint cdf* is defined $F_{X,Y}(x,y) = \mathbf{Pr}(X \leq x, Y \leq y)$. The *marginal cumulative distribution functions* of $F_{X,Y}$ are defined as $F_X(x) = \lim_{y \to \infty} F_{X,Y}(x,y)\mathrm{d}y$ and $F_Y(y) = \lim_{x \to \infty} F_{X,Y}(x,y)\mathrm{d}x$.

Similarly, when Y is discrete, the *marginal pdf* is defined $f_X(x) = \sum_{y \in \Omega_Y} f_{X,Y}(x,y) = \sum_{y \in \Omega_Y} \mathbf{Pr}(X = x, Y = y)$. When the random variables are continuous, we define $f_{X,Y}(x,y) = \frac{\mathrm{d}^2 F_{X,Y}(x,y)}{\mathrm{d}x\mathrm{d}y}$. And then the marginal pdf of X (when $\Omega_Y = \mathbb{R}$) is defined $f_X(x) = \int_{y=-\infty}^{\infty} f_{X,Y}(x,y)\mathrm{d}y$. Marginalizing removes the effect of a random variable (Y in the above definitions).

Now we can say random variables X and Y are independent if and only if $f_{X,Y}(x,y) = f_X(x) \cdot f_Y(y)$ for all x and y.

Then a *conditional distribution* of X given $Y = y$ is defined $f_{X|Y}(x \mid y) = f_{X,Y}(x,y)/f_Y(y)$ (given that $f_Y(y) \neq 0$).

 copyright: Jeff M. Phillips

Example: Marginal and Conditional Distributions

Consider someone who randomly chooses his pants and shirt every day (a to-remain-anonymous friend of the author's actually did this in college – all clothes were in a pile, clean or dirty; when the average article of clothing was too smelly, all were washed). Let P be a random variable for the color of pants, and S a random variable for the color of the shirt. Their joint probability is described by this table:

	S=green	S=red	S=blue
P=blue	0.3	0.1	0.2
P=white	0.05	0.2	0.15

Adding up along columns, the marginal distribution f_S for the color of the shirt is described by the following table:

S=green	S=red	S=blue
0.35	0.3	0.35

Isolating and renormalizing the middle "S=red" column, the conditional distribution $f_{P|S}(\cdot \mid S\text{= red})$ is described by the following table:

P=blue	P=white
$\frac{0.1}{0.3} = 0.3333$	$\frac{0.2}{0.3} = 0.6666$

Example: Gaussian Distribution

The Gaussian distribution is a d-variate distribution $N_d : \mathbb{R}^d \to \mathbb{R}$ that generalizes the one-dimensional normal distribution. The definition of the symmetric version (we will generalize to non-trivial covariance later on) depends on a mean $\mu \in \mathbb{R}^d$ and a variance σ^2. For any vector $v \in \mathbb{R}^d$, it is defined

$$N_d(v) = \frac{1}{\sigma^d \sqrt{2\pi^d}} \exp\left(-\|v - \mu\|^2/(2\sigma^2)\right).$$

For the 2-dimensional case where $v = (v_x, v_y)$ and $\mu = (\mu_x, \mu_y)$, then this is defined

$$N_2(v) = \frac{1}{\sigma^2 \pi \sqrt{2}} \exp\left(-((v_x - \mu_x)^2 - (v_y - \mu_y)^2)/(2\sigma^2)\right).$$

A *magical* property about the Gaussian distribution is that all conditional versions of it are also Gaussian, of a lower dimension. For instance, in the two dimensional case $N_2(v_x \mid v_y = 1)$ is a 1-dimensional Gaussian, or a normal distribution. There are many other essential properties of the Gaussian that we will see throughout this text, including that it is invariant under all basis transformations and that it is the limiting distribution for central limit theorem bounds.

copyright: Jeff M. Phillips

Exercises

Q1.1: Consider the probability table below for the random variables X and Y. One entry is missing, but you should be able to derive it. Then calculate the following values.

1. **Pr**$(X = 3 \cap Y = 2)$
2. **Pr**$(Y = 1)$
3. **Pr**$(X = 2 \cap Y = 1)$
4. **Pr**$(X = 2 \mid Y = 1)$

	$X = 1$	$X = 2$	$X = 3$
$Y = 1$	0.25	0.1	0.15
$Y = 2$	0.1	0.2	??

Q1.2: An "adventurous" athlete has the following running routine every morning: He takes a bus to a random stop, then hitches a ride, and then runs all the way home. The bus, described by a random variable B, has four stops where the stops are at a distance of 1, 3, 4, and 7 miles from his house – he chooses each with probability $1/4$. Then the random hitchhiking takes him further from his house with a uniform distribution between -1 and 4 miles; that is it is represented as a random variable H with pdf described

$$f(H = x) = \begin{cases} 1/5 & \text{if } x \in [-1, 4] \\ 0 & \text{if } x \notin [-1, 4]. \end{cases}$$

What is the expected distance he runs each morning (all the way home)?

Q1.3: Consider rolling two fair die D_1 and D_2; each has a probability space of $\Omega = \{1, 2, 3, 4, 5, 6\}$ which each value equally likely. What is the probability that D_1 has a larger value than D_2? What is the expected value of the sum of the two die?

Q1.4: Let X be a random variable with a uniform distribution over $[0, 2]$; its pdf is described

$$f(X = x) = \begin{cases} 1/2 & \text{if } x \in [0, 2] \\ 0 & \text{if } x \notin [0, 2]. \end{cases}$$

What is the probability that $f(X = 1)$?

Q1.5: Use python to plot the pdf and cdf of the Laplace distribution ($f(x) = \frac{1}{2}\exp(-|x|)$) for values of x in the range $[-3, 3]$. The function `scipy.stats.laplace` may be useful.

Q1.6: Consider the random variables X and Y described by the joint probability table.

	$X = 1$	$X = 2$	$X = 3$
$Y = 1$	0.10	0.05	0.10
$Y = 2$	0.30	0.25	0.20

Derive the following values

1. **Pr**$(X = 1)$
2. **Pr**$(X = 2 \cap Y = 1)$
3. **Pr**$(X = 3 \mid Y = 2)$

copyright: Jeff M. Phillips

Compute the following probability distributions.

4. What is the marginal distribution for X?
5. What is the conditional probability for Y, given that $X = 2$?

Answer the following question about the joint distribution.

6. Are random variables X and Y independent?
7. Is $\mathbf{Pr}(X = 1)$ independent of $\mathbf{Pr}(Y = 1)$?

copyright: Jeff M. Phillips

2 Bayes' Rule

This topic is on Bayes' Rule and Bayesian Reasoning. Bayes' Rule is the key component in how to build likelihood functions, which when optimized are key to evaluating "models" based on data. Bayesian Reasoning is a much broader area that can go well beyond just finding the single "most optimal" model. This line of work, which this chapter will only introduce, reasons about the many possible models and can make predictions with this uncertainty taken into account.

2.1 Bayes' Rule

Given two events M and D Bayes' Rule states

$$\textbf{Pr}(M \mid D) = \frac{\textbf{Pr}(D \mid M) \cdot \textbf{Pr}(M)}{\textbf{Pr}(D)}.$$

Mechanically, this provides an algebraic way to invert the direction of the conditioning of random variables, from (D given M) to (M given D). It assumes nothing about the independence of M and D (otherwise its pretty uninteresting). To derive this we use

$$\textbf{Pr}(M \cap D) = \textbf{Pr}(M \mid D)\textbf{Pr}(D)$$

and also

$$\textbf{Pr}(M \cap D) = \textbf{Pr}(D \cap M) = \textbf{Pr}(D \mid M)\textbf{Pr}(M).$$

Combining these we obtain $\textbf{Pr}(M \mid D)\textbf{Pr}(D) = \textbf{Pr}(D \mid M)\textbf{Pr}(M)$, from which we can divide by $\textbf{Pr}(D)$ to solve for $\textbf{Pr}(M \mid D)$. So Bayes' Rule is uncontroversially true; any "frequentist vs. Bayesian" debate is about how to model data and perform analysis, not the specifics or correctness of this rule.

Example: Checking Bayes' Rule

Consider two events M and D with the following joint probability table:

	$M = 1$	$M = 0$
$D = 1$	0.25	0.5
$D = 0$	0.2	0.05

We can observe that indeed $\textbf{Pr}(M \mid D) = \textbf{Pr}(M \cap D)/\textbf{Pr}(D) = \frac{0.25}{0.75} = \frac{1}{3}$, which is equal to

$$\frac{\textbf{Pr}(D \mid M)\textbf{Pr}(M)}{\textbf{Pr}(D)} = \frac{\frac{.25}{.2+.25}(.2 + .25)}{.25 + .5} = \frac{.25}{.75} = \frac{1}{3}.$$

But Bayes' rule is not very interesting in the above example. In that example, it is actually *more* complicated to calculate the right side of Bayes' rule than it is the left side.

Example: Cracked Windshield

Consider you bought a new car and its windshield was cracked, the event W. If the car was assembled at one of three factories A, B or C, you would like to know which factory was the most likely point of origin.

Assume that in Utah 50% of cars are from factory A (that is $\mathbf{Pr}(A) = 0.5$) and 30% are from factory B ($\mathbf{Pr}(B) = 0.3$), and 20% are from factory C ($\mathbf{Pr}(C) = 0.2$).

Then you look up statistics online, and find the following rates of cracked windshields for each factory – apparently this is a problem! In factory A, only 1% are cracked, in factory B 10% are cracked, and in factory C 2% are cracked. That is $\mathbf{Pr}(W \mid A) = 0.01$, $\mathbf{Pr}(W \mid B) = 0.1$ and $\mathbf{Pr}(W \mid C) = 0.02$.

We can now calculate the probability the car came from each factory:

- $\mathbf{Pr}(A \mid W) = \mathbf{Pr}(W \mid A) \cdot \mathbf{Pr}(A)/\mathbf{Pr}(W) = 0.01 \cdot 0.5/\mathbf{Pr}(W) = 0.005/\mathbf{Pr}(W)$.
- $\mathbf{Pr}(B \mid W) = \mathbf{Pr}(W \mid B) \cdot \mathbf{Pr}(B)/\mathbf{Pr}(W) = 0.1 \cdot 0.3/\mathbf{Pr}(W) = 0.03/\mathbf{Pr}(W)$.
- $\mathbf{Pr}(C \mid W) = \mathbf{Pr}(W \mid C) \cdot \mathbf{Pr}(C)/\mathbf{Pr}(W) = 0.02 \cdot 0.2/\mathbf{Pr}(W) = 0.004/\mathbf{Pr}(W)$.

We did not calculate $\mathbf{Pr}(W)$, but it must be the same for all factory events, so to find the highest probability factory we can ignore it. The probability $\mathbf{Pr}(B \mid W) = 0.03/\mathbf{Pr}(W)$ is the largest, and B is the most likely factory.

2.1.1 Model Given Data

In data analysis, M represents a 'model' and D as 'data.' Then $\mathbf{Pr}(M \mid D)$ is interpreted as the probability of model M given that we have observed D. A *maximum a posteriori* (or MAP) estimate is the model $M \in \Omega_M$ that maximizes $\mathbf{Pr}(M \mid D)$. That is[1]

$$M^* = \operatorname*{argmax}_{M \in \Omega_M} \mathbf{Pr}(M \mid D) = \operatorname*{argmax}_{M \in \Omega_M} \frac{\mathbf{Pr}(D \mid M)\mathbf{Pr}(M)}{\mathbf{Pr}(D)} = \operatorname*{argmax}_{M \in \Omega_M} \mathbf{Pr}(D \mid M)\mathbf{Pr}(M).$$

Thus, by using Bayes' Rule, we can maximize $\mathbf{Pr}(M \mid D)$ using $\mathbf{Pr}(M)$ and $\mathbf{Pr}(M \mid D)$. We do not need $\mathbf{Pr}(D)$ since our data is given to us and fixed for all models.

In some settings we may also ignore $\mathbf{Pr}(M)$, as we may assume all possible models are equally likely. This is not always the case, and we will come back to this. In this setting we just need to calculate $\mathbf{Pr}(D \mid M)$. This function $L(M) = \mathbf{Pr}(D \mid M)$ is called the *likelihood* of model M.

So what is a 'model' and what is 'data?' A *model* is usually a simple pattern from which we think data is generated, but then observed with some noise. Examples:

- The model M is a single point in $\mathbb{R}^d$; the data is a set of points in $\mathbb{R}^d$ near M.
- **linear regression:** The model M is a line in $\mathbb{R}^2$; the data is a set of points such that for each x-coordinate, the y-coordinate is the value of the line at that x-coordinate with some added noise in the y-value.
- **clustering:** The model M is a small set of points in $\mathbb{R}^d$; the data is a large set of points in $\mathbb{R}^d$, where each point is near one of the points in M.

[1] Consider a set S, and a function $f : S \to \mathbb{R}$. The $\max_{s \in S} f(s)$ returns the *value* $f(s^*)$ for some element $s^* \in S$ which results in the largest valued $f(s)$. The $\operatorname{argmax}_{s \in S} f(s)$ returns the *element* $s^* \in S$ which results in the largest valued $f(s)$; if this is not unique, it may return any such $s^* \in S$.

- **PCA:** The model M is a k-dimensional subspace in $\mathbb{R}^d$ (for $k \ll d$); the data is a set of points in $\mathbb{R}^d$, where each point is near M.
- **linear classification:** The model M is a halfspace in $\mathbb{R}^d$; the data is a set of labeled points (with labels $+$ or $-$), so the $+$ points are mostly in M, and the $-$ points are mainly not in M.

Log-likelihoods. An important trick used in understanding the likelihood, and in finding the MAP model M^*, is to take the logarithm of the posterior. Since the logarithm operator $\log(\cdot)$ is monotonically increasing on positive values, and all probabilities (and more generally pdf values) are non-negative (treat $\log(0)$ as $-\infty$), then $\text{argmax}_{M\in\Omega_M} \textbf{Pr}(M \mid D) = \text{argmax}_{M\in\Omega_M} \log(\textbf{Pr}(M \mid D))$. It is commonly applied on only the likelihood function $L(M)$, and $\log(L(M))$ is called the log-likelihood. Since $\log(a\cdot b) = \log(a)+\log(b)$, this is useful in transforming definitions of probabilities, which are often written as products $\Pi_{i=1}^k P_i$ into sums $\log(\Pi_{i=1}^k P_i) = \sum_{i=1}^k \log(P_i)$, which are easier to manipulated algebraically.

Moreover, the base of the log is unimportant in model selection using the MAP estimate because $\log_{b_1}(x) = \log_{b_2}(x)/\log_{b_2}(b_1)$, and so $1/\log_{b_2}(b_1)$ is a coefficient that does not affect the choice of M^*. The same is true for the *maximum likelihood estimate (MLE):* $M^* = \text{argmax}_{M\in\Omega_M} L(M)$.

Example: Gaussian MLE

Let the data D be a set of points in $\mathbb{R}^1$: $\{1, 3, 12, 5, 9\}$. Let Ω_M be $\mathbb{R}$ so that the model is parametrized by a point $M \in \mathbb{R}$. If we assume that each data point is observed with independent Gaussian noise (with $\sigma = 2$, so its pdf is described as $g(x) = \frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(M-x)^2)$. Then

$$\textbf{Pr}(D \mid M) = \prod_{x\in D} g(x) = \prod_{x\in D} \left(\frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(M-x)^2)\right).$$

Recall that we can take the product $\Pi_{x\in D} g(x)$ since we assume independence of $x \in D$! To find $M^* = \text{argmax}_M \textbf{Pr}(D \mid M)$ is equivalent to $\text{argmax}_M \ln(\textbf{Pr}(D \mid M))$, the *log-likelihood* which is

$$\ln(\textbf{Pr}(D \mid M)) = \ln\left(\prod_{x\in D}\left(\frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(M-x)^2)\right)\right) = \sum_{x\in D}\left(-\frac{1}{8}(M-x)^2\right) + |D| \ln(\frac{1}{\sqrt{8\pi}}).$$

We can ignore the last term in the sum since it does not depend on M. The first term is maximized when $\sum_{x\in D}(M-x)^2$ is minimized, which occurs precisely as $\textbf{E}[D] = \frac{1}{|D|}\sum_{x\in D} x$, the mean of the data set D. That is, the maximum likelihood model is exactly the mean of the data D, and is quite easy to calculate.

2.2 Bayesian Inference

Bayesian inference focuses on a simplified version of Bayes's Rule:

$$\textbf{Pr}(M \mid D) \propto \textbf{Pr}(D \mid M) \cdot \textbf{Pr}(M).$$

The symbol $\propto$ means *proportional to*; that is there is a fixed (but possibly unknown) constant factor c multiplied on the right (in this case $c = 1/\textbf{Pr}(D)$) to make them equal: $\textbf{Pr}(M \mid D) = c\cdot\textbf{Pr}(D \mid M)\cdot\textbf{Pr}(M)$.

However, we may want to use continuous random variables, so then strictly using probability **Pr** at a single point is not always correct. So we can replace each of these with pdfs

$$\underset{\text{posterior}}{p(M \mid D)} \quad \propto \quad \underset{\text{likelihood}}{f(D \mid M)} \quad \cdot \quad \underset{\text{prior}}{\pi(M)}.$$

 copyright: Jeff M. Phillips

Each of these terms have common names. As above, the conditional probability or pdf $\mathbf{Pr}(D \mid M) \propto f(D \mid M)$ is called the *likelihood*; it evaluates the effectiveness of the model M using the observed data D. The probability or pdf of the model $\mathbf{Pr}(M) \propto \pi(M)$ is called the *prior*; it is the assumption about the relative propensity of a model M, before or independent of the observed data. And the left hand side $\mathbf{Pr}(M \mid D) \propto p(M \mid D)$ is called the *posterior*; it is the combined evaluation of the model that incorporates how well the observed data fits the model and the independent assumptions about the model.

Again it is common to be in a situation where, given a fixed model M, it is possible to calculate the likelihood $f(D \mid M)$. And again, the goal is to be able to compute $p(M \mid D)$, as this allows us to evaluate potential models M, given the data we have seen D.

The main difference is a careful analysis of $\pi(M)$, the prior – which is not necessarily assumed uniform or "flat". The prior allows us to encode our assumptions.

Example: Average Height

Lets estimate the height H of a typical Data University student. We can construct a data set $D = \{x_1, \ldots, x_n\}$ by measuring the height of everyone in this class in inches. There may be error in the measurement, and we are an incomplete set, so we don't entirely trust the data.
So we introduce a prior $\pi(M)$. Consider we read that the average height of a full grown person is $\mu_M = 66$ inches, with a standard deviation of $\sigma = 6$ inches. So we assume

$$\pi(M) = N(66, 6) = \frac{1}{\sqrt{\pi 72}} \exp(-(\mu_M - 66)^2/(2 \cdot 6^2)),$$

is normally distributed around 66 inches.
Now, given this knowledge we adjust the MLE example from last subsection using this prior.

- *What if our MLE estimate without the prior (e.g. $\frac{1}{|D|}\sum_{x \in D} x$) provides a value of 55?*
  The data is very far from the prior! Usually this means something is wrong. We could find $\text{argmax}_M\, p(M \mid D)$ using this information, but that may give us an estimate of say 20 (that does not seem correct). A more likely explanation is a mistake somewhere: probably we measured in feet instead of inches!

Another vestige of Bayesian inference is that we not only can calculate the maximum likelihood model M^*, but we can also provide a posterior value for any model! This value is not an absolute probability (its not normalized, and regardless it may be of measure 0), but it is powerful in other ways:

- We can say (under our model assumptions, which are now clearly stated) that one model M_1 is twice as likely as another M_2, if $p(M_1 \mid D)/p(M_2 \mid D) = 2$.

- We can define a range of parameter values (with more work and under our model assumptions) that likely contains the true model.

- We can now use more than one model for prediction of a value. Given a new data point x' we may want to map it onto our model as $M(x')$, or assign it a score of fit. Instead of doing this for just one "best" model M^*, we can take a weighted average of all models, weighted by their posterior; this is "marginalization."

Weight for Prior. So how important is the prior? In the average height example, it will turn out to be worth only $(1/9)$th of one student's measurement. But we can give it more weight.

Example: Weighted Prior for Height

Lets continue the example about the height of an average Data University student, and assume (as in the MLE example) the data is generated independently from a model M with Gaussian noise with $\sigma = 2$. Thus the likelihood of the model, given the data is

$$f(D \mid M) = \prod_{x \in D} g(x) = \prod_{x \in D} \left(\frac{1}{\sqrt{8\pi}} \exp(-\frac{1}{8}(\mu_M - x)^2) \right).$$

Now using that the prior of the model is $\pi(M) = \frac{1}{\sqrt{\pi 72}} \exp(-(\mu_M - 66)^2/72)$, the posterior is given by

$$p(M \mid D) \propto f(D \mid M) \cdot \frac{1}{\sqrt{\pi 72}} \exp(-(\mu_M - 66)^2/72).$$

It is again easier to work with the log-posterior which is monotonic with the posterior, using some unspecified constant C (which can be effectively ignored):

$$\begin{aligned} \ln(p(M \mid D)) &\propto \ln(f(D \mid M)) + \ln(\pi(M)) + C \\ &\propto \sum_{x \in D} \left(-\frac{1}{8}(\mu_M - x)^2) \right) - \frac{1}{72}(\mu_M - 66)^2 + C \\ &\propto -\sum_{x \in D} 9(\mu_M - x)^2 + (\mu_M - 66)^2 + C \end{aligned}$$

So the maximum likelihood estimator occurs at the average of 66 along with 9 copies of the student data.

Why is student measurement data worth so much more?

We assume the standard deviation of the measurement error is 2, where as we assumed that the standard deviation of the full population was 6. In other words, our measurements had variance $2^2 = 4$, and the population had variance $6^2 = 36$ (technically, this is best to interpret as the variance when adapted to various subpopulations, e.g., Data University students): that is 9 times as much.

If instead we assumed that the standard deviation of our prior is 0.1, with variance 0.01, then this is 400 times smaller than our class measurement error variance. If we were to redo the above calculations with this smaller variance, we would find that this assumption weights the prior 400 times the effect of each student measurement in the MLE.

In fact, a much smaller variance on the prior is probably more realistic since national estimates on height are probably drawn from a very large sample. And its important to keep in mind that we are estimating the *average height* of a population, not the *height of a single person* randomly drawn from the population. In the next topic (T3) we will see how averages of random variables have much smaller variance – are much more concentrated – than individual random variables.

So what happens with more data?

Lets say, this class gets really popular, and next year $1{,}000$ or $10{,}000$ students sign up! Then again the student data is overall worth more than the prior data. So with any prior, if we get enough data, it no longer becomes important. But with a small amount of data, it can have a large influence on our model.

copyright: Jeff M. Phillips

Exercises

Q2.1: Consider two models M_1 and M_2, where from prior knowledge we believe that $\textbf{Pr}(M_1) = 0.25$ and $\textbf{Pr}(M_2) = 0.75$. We then observe a data set D. Given each model we assess the likelihood of seeing that data given the model as $\textbf{Pr}(D \mid M_1) = 0.5$ and $\textbf{Pr}(D \mid M_2) = 0.01$. Now that we have the data, which model is has a higher probability of being correct?

Q2.2: Assume I observe 3 data points x_1, x_2, and x_3 drawn independently from an unknown distribution. Given a model M, I can calculate the likelihood for each data point as $\textbf{Pr}(x_1 \mid M) = 0.5$, $\textbf{Pr}(x_2 \mid M) = 0.1$, and $\textbf{Pr}(x_3 \mid M) = 0.2$. What is the likelihood of seeing all of these data points, given the model M: $\textbf{Pr}(x_1, x_2, x_3 \mid M)$?

Q2.3: Consider a data set D with 10 data points $\{-1, 6, 0, 2, -1, 7, 7, 8, 4, -2\}$. We want to find a model for M from a restricted sample space $\Omega = \{0, 2, 4\}$. Assume the data has Laplace noise defined, so from a model M a data point's probability distribution is described $f(x) = \frac{1}{4}\exp(-|M - x|/2)$. Also assume we have a prior assumption on the models so that $\textbf{Pr}(M = 0) = 0.25$, $\textbf{Pr}(M = 2) = 0.35$, and $\textbf{Pr}(M = 4) = 0.4$. Assuming all data points in D are independent, which model is most likely.

copyright: Jeff M. Phillips

3 Convergence

This topic will overview a variety of extremely powerful analysis results that span statistics, estimation theorem, and big data. It provides a framework to think about how to aggregate more and more data to get better and better estimates. It will cover the *Central Limit Theorem* (CLT), Chernoff-Hoeffding bounds, and Probably Approximately Correct (PAC) algorithms.

3.1 Sampling and Estimation

Most data analysis starts with some data set; we will call this data set P. It will be composed of a set of n data points $P = \{p_1, p_2, \ldots, p_n\}$.

But underlying this data is almost always a very powerful assumption, that this data comes iid from a fixed, but usually unknown pdf, call this f. Lets unpack this: What does "iid" mean: Identically and Independently Distributed. The "identically" means each data point was drawn from the same f. The "independently" means that the first points have no bearing on the value of the next point.

Example: Polling

Consider a poll of $n = 1000$ likely voters in an upcoming election. If we assume each polled person is chosen iid, then we can use this to understand something about the underlying distribution f, for instance the distribution of all likely voters.
More generally, f could represent the outcome of a process, whether that is a randomized algorithm, a noisy sensing methodology, or the common behavior of a species of animals. In each of these cases, we essentially "poll" the process (algorithm, measurement, thorough observation) having it provide a sample, and repeat many times over.

Here we will talk about estimating the mean of f. To discuss this, we will now introduce a random variable $X \sim f$; a hypothetical new data point. The *mean* of f is the expected value of X: $\mathbf{E}[X]$.

We will estimate the mean of f using the *sample mean*, defined $\bar{P} = \frac{1}{n}\sum_{i=1}^{n} p_i$. The following diagram represents this common process: from a unknown process f, we consider n iid random variables $\{X_i\}$ corresponding to a set of n independent observations $\{p_i\}$, and take their average $\bar{P} = \frac{1}{n}\sum_{i=1}^{n} p_i$ to estimate the mean of f.

$$\bar{P} \underset{\frac{1}{n}\sum}{=} \{p_i\} \underset{\text{realize}}{\leftarrow} \{X_i\} \underset{\text{iid}}{\sim} f$$

Central Limit Theorem. The central limit theorem is about how well the sample mean approximates the true mean. But to discuss the sample mean $\bar{P}$ (which is a fixed value) we need to discuss random variables $\{X_1, X_2, \ldots, X_n\}$, and their mean $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. Note that again $\bar{X}$ is a random variable. If we are to draw a new iid data set P' and calculate a new sample mean $\bar{P}'$ it will likely not be exactly the same as $\bar{P}$; however, the distribution of where this $\bar{P}'$ is likely to be, is precisely $\bar{X}$. Arguably, this distribution is more important than $\bar{P}$ itself.

There are many formal variants of the central limit theorem, but the basic form is as follows:

Central Limit Theorem: Consider n iid random variables $X_1, X_2, \ldots, X_n$, where each $X_i \sim f$ for any fixed distribution f with mean μ and bounded variance σ^2. Then $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ converges to the normal distribution with mean $\mu = \mathbf{E}[X_i]$ and variance σ^2/n.

Lets highlight some important consequences:

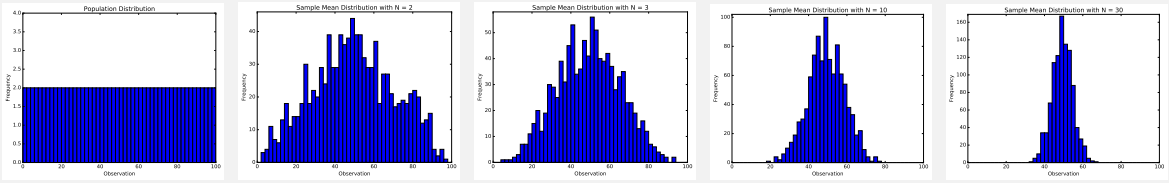
- For any f (that is not too crazy, since σ^2 is not infinite), then $\bar{X}$ looks like a normal distribution.
- The mean of the normal distribution, which is the expected value of $\bar{X}$ satisfies $\mathbf{E}[\bar{X}] = \mu$, the mean of f. This implies that we can use $\bar{X}$ (and then also $\bar{P}$) as a guess for μ.
- As n gets larger (we have more data points) then the variance of $\bar{X}$ (our estimator) decreases. So keeping in mind that although $\bar{X}$ has the right expected value it also has some error, this error is decreasing as n increases.

```
# adapted from:  https://github.com/mattnedrich/CentralLimitTheoremDemo
import random
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt

def plot_distribution(distribution, file, title, bin_min, bin_max, num_bins):
    bin_size = (bin_max - bin_min) / num_bins
    manual_bins = range(bin_min, bin_max + bin_size, bin_size)
    [n, bins, patches] = plt.hist(distribution, bins = manual_bins)
    plt.title(title)
    plt.xlim(bin_min, bin_max)
    plt.ylim(0, max(n) + 2)
    plt.ylabel("Frequency")
    plt.xlabel("Observation")
    plt.savefig(file, bbox_inches='tight')
    plt.clf()
    plt.cla()


minbin = 0
maxbin = 100
numbins = 50
nTrials = 1000

def create_uniform_sample_distribution():
    return range(maxbin)
sampleDistribution = create_uniform_sample_distribution()

# Plot the original population distribution
plot_distribution(sampleDistribution, 'output/SampleDistribution.pdf',
  "Population Distribution", minbin, maxbin, numbins)

# Plot a sampling distribution for values of N = 2, 3, 10, and 30
n_vals = [2, 3, 10, 30]
for N in n_vals:
  means = []
  for j in range(nTrials):
    sampleSum = 0;
    for i in range(N):
      sampleSum += random.choice(sampleDistribution)
```

```
    means.append(float(sampleSum)/ float(N))

title = "Sample Mean Distribution with N = %s" % N
file = "output/CLT-demo-%s.pdf" % N
plot_distribution(means, file, title, minbin, maxbin, numbins)
```

Example: Central Limit Theorem

Consider f as a uniform distribution over $[0, 100]$. If we create n samples $\{p_1, \ldots, p_n\}$ and their mean $\bar{P}$, then repeat this 1000 times, we can plot the output in histograms:



We see that starting at $n = 2$, the distributions look vaguely looks normal (in the technical sense of a normal distribution), and that their standard deviations narrow as n increases.

Remaining Mysteries. There should still be at least a few aspects of this not clear yet: (1) What does "convergence" mean? (2) How do we formalize or talk about this notion of error? (3) What does this say about our data $\bar{P}$?

First, *convergence* refers to what happens as some parameter increases, in this case n. As the number of data points increase, as n "goes to infinity" then the above statement ($\bar{X}$ looks like a normal distribution) becomes more and more true. For small n, the distribution may not quite look like a normal, it may be more bumpy, maybe even multi-modal. The statistical definitions of convergence are varied, and we will not go into them here, we will instead replace it with more useful phrasing in explaining aspects (2) and (3).

Second, the error now has two components. We cannot simply say that $\bar{P}$ is at most some distance ε from μ. Something crazy might have happened (the sample is random after all). And it is not useful to try to write the probability that $\bar{P} = \mu$; for equality in continuous distributions, this probability is indeed 0. But we can combine these notions. We can say the distance between $\bar{P}$ and μ is more than ε, with probability at most δ. This is called "probably approximately correct" or PAC.

Third, we want to generate some sort of PAC bound (which is far more useful than "$\bar{X}$ looks kind of like a normal distribution"). Whereas a frequentist may be happy with a confidence interval and a Bayesian a normal posterior, these two options are not directly available since again, $\bar{X}$ is not exactly a normal. So we will discuss some very common *concentration of measure* tools. These don't exactly capture the shape of the normal distribution, but provide upper bounds for its tails, and will allow us to state PAC bounds.

3.2 Probably Approximately Correct (PAC)

We will introduce shortly the three most common concentration of measure bounds, which provide increasingly strong bounds on the tails of distributions, but require more and more information about the underlying distribution f. Each provides a PAC bound of the following form:

$$\mathbf{Pr}[|\bar{X} - \mathbf{E}[\bar{X}]| \geq \varepsilon] \leq \delta.$$

That is, the probability that $\bar{X}$ (which is some random variable, often a sum of iid random variables) is further than ε to its expected value (which is μ, the expected value of f where $X_i \sim f$), is at most δ. Note we do not try to say this probability is *exactly* δ, this is often too hard. In practice there are a variety of tools, and a user may try each one, and see which ones gives the best bound.

It is useful to think of ε as the *error tolerance* and δ as the *probability of failure*, i.e., that we exceed the error tolerance. However, often these bounds will allow us to write the required sample size n in terms of ε and δ. This allows us to trade these two terms off for any fixed known n; we can gaurantee a smaller error tolerance if we are willing to allow more probability of failure, and vice-versa.

3.3 Concentration of Measure

We will formally describe these bounds, and give some intuition of why they are true (but not proofs). But what will be the most important is what they imply. If you just know the distance of the expectation from the minimal value, you can get a very weak bound. If you know the variance of the data, you can get a stronger bound. If you know that the distribution f has a small and bounded range, then you can make the probability of failure (the δ in PAC bounds) very very small.

Markov Inequality. Let X be a random variable such that $X \geq 0$, that is it cannot take on negative values. Then for any parameter $\alpha > 0$

$$\mathbf{Pr}[X > \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}.$$

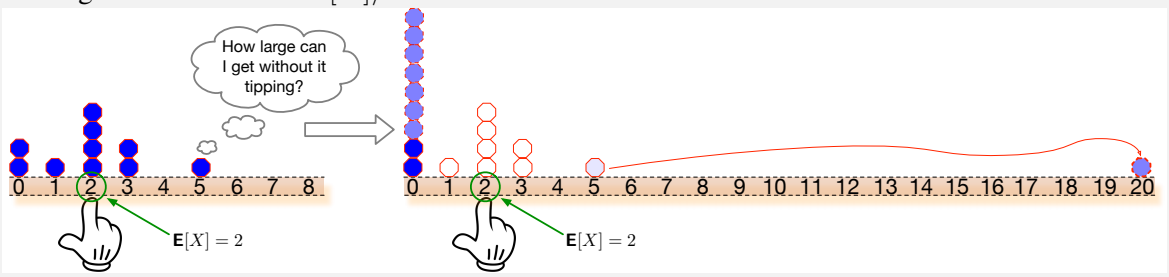
Note this is a PAC bound with $\varepsilon = \alpha - \mathbf{E}[X]$ and $\delta = \mathbf{E}[X]/\alpha$, or we can rephrase this bound as follows: $\mathbf{Pr}[X - \mathbf{E}[X] > \varepsilon] \leq \delta = \mathbf{E}[X]/(\varepsilon + \mathbf{E}[X])$.

Geometry of the Markov Inequality

Consider balancing the pdf of some random variable X on your finger at $\mathbf{E}[X]$, like a waitress balances a tray. If your finger is not under a value μ so $\mathbf{E}[X] = \mu$, then the pdf (and the waitress's tray) will tip, and fall in the direction of μ – the "center of mass."
Now for some amount of probability α, how large can we increase its location so we retain $\mathbf{E}[X] = \mu$. For each part of the pdf we increase, we must decrease some in proportion. However, by the assumption $X \geq 0$, the pdf must not be positive below 0. In the limit of this, we can set $\mathbf{Pr}[X = 0] = 1 - \alpha$, and then move the remaining α probability as large as possible, to a location δ so $\mathbf{E}[X] = \mu$. That is

$$\mathbf{E}[X] = 0 \cdot \mathbf{Pr}[X = 0] + \delta \cdot \mathbf{Pr}[X = \delta] = 0 \cdot (1 - \alpha) + \delta \cdot \alpha = \delta \cdot \alpha.$$

Solving for δ we find $\delta = \mathbf{E}[X]/\alpha$.



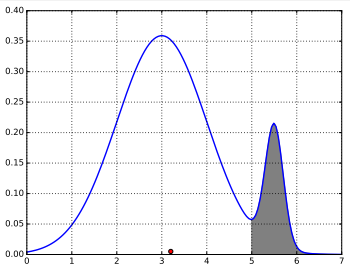
Imagine having 10 α-balls each representing $\alpha = 1/10$th of the probability mass. As in the figure, if these represent a distribution with $\mathbf{E}[X] = 2$ and this must stay fixed, how far can one ball increase if all others balls must take a value at least 0? One ball can move to 20.

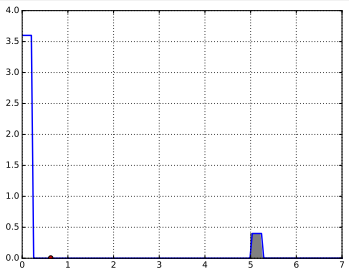
If we instead know that $X \geq b$ for some constant b (instead of $X \geq 0$), then we state more generally $\mathbf{Pr}[X > \alpha] \leq (\mathbf{E}[X] - b)/(\alpha - b)$.

copyright: Jeff M. Phillips

Example: Markov Inequality

Consider the pdf f drawn in blue in the following figures, with $\mathbf{E}[X]$ for $X \sim f$ marked as a red dot. The probability that X is greater than 5 (e.g. $\mathbf{Pr}[X \geq 5]$) is the shaded area.





Notice that in both cases that $\mathbf{Pr}[X \geq 5]$ is about 0.1. This is the quantity we want to bound by above by δ. But since $\mathbf{E}[X]$ is much larger in the first case (about 2.25), then the bound $\delta = \mathbf{E}[X]/\alpha$ is much larger, about 0.45. In the second case, $\mathbf{E}[X]$ is much smaller (about 0.6) so we get a much better bound of $\delta = 0.12$.

Example: Markov Inequality and Coffee

Let C be a random variable describing the number liters of coffee the faculty at Data University will drink in a week. Say we know $\mathbf{E}[C] = 20$.
We use the Markov Inequality to bound the probability that the coffee consumed will be more than 50 liters as

$$\mathbf{Pr}[C \geq 50] \leq \frac{\mathbf{E}[C]}{50} = \frac{20}{50} = 0.4$$

Hence, based on the expected value alone, we can bound the with probability less than 0.4, the faculty at DU will drink less than 50 liters of coffee.

Chebyshev Inequality. Now let X be a random variable where we know $\mathbf{Var}[X]$, and $\mathbf{E}[X]$. Then for any parameter $\varepsilon > 0$

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq \varepsilon] \leq \frac{\mathbf{Var}[X]}{\varepsilon^2}.$$

Again, this clearly is a PAC bound with $\delta = \mathbf{Var}[X]/\varepsilon^2$. This bound is typically stronger than the Markov one since δ decreases quadratically in ε instead of linearly.

 copyright: Jeff M. Phillips

Example: Chebyshev Inequality and Coffee

Again let C be a random variable for the liters of coffee that faculty at Data University will drink in a week with $\mathbf{E}[C] = 20$. If we also know that the variance is not too large, specifically $\mathbf{Var}[C] = 9$ (liters squared), then we can apply the Chebyshev inequality to get an improved bound.

$$\mathbf{Pr}[C \geq 50] \leq \mathbf{Pr}[|C - \mathbf{E}[C]| \geq 30] \leq \frac{\mathbf{Var}[C]}{30^2} = \frac{9}{900} = 0.01$$

That is, by using the expectation ($\mathbf{E}[C] = 20$) and variance ($\mathbf{Var}[C] = 9$) we can reduce the probability of exceeding 50 liters to at most probability 0.01.
Note that in the first inequality we convert from a one-sided expression $C \geq 50$ to a two-sided expression $|C - \mathbf{E}[C]| \geq 30$ (that is either $C - \mathbf{E}[C] \geq 30$ or $\mathbf{E}[C] - C \geq 30$). This is a bit wasteful, and stricter one-sided variants of Chebyshev inequality exist; we will not discuss these here in an effort for simplicity.

Recall that for an average of random variables $\bar{X} = (X_1 + X_2 + \ldots + X_n)/n$, where the X_is are iid, and have variance σ^2, then $\mathbf{Var}[\bar{X}] = \sigma^2/n$. Hence

$$\mathbf{Pr}[|\bar{X} - \mathbf{E}[X_i]| \geq \varepsilon] \leq \frac{\sigma^2}{n\varepsilon^2}.$$

Consider now that we have input parameters ε and δ, our desired error tolerance and probability of failure. If can draw $X_i \sim f$ (iid) for an unknown f (with known expected value and variance σ), then we can solve for how large n needs to be: $n = \sigma^2/(\varepsilon^2\delta)$.

Since $\mathbf{E}[\bar{X}] = \mathbf{E}[X_i]$ for iid random variables $X_1, X_2, \ldots, X_n$, there is not a similar meaningfully-different extension for the Markov inequality.

Chernoff-Hoeffding Inequality. Following the above extension of the Chebyshev inequality, we can consider a set of n iid random variables $X_1, X_2, \ldots, X_n$ where $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. Now assume we know that each X_i lies in a bounded domain $[b, t]$, and let $\Delta = t - b$. Then for any parameter $\varepsilon > 0$

$$\mathbf{Pr}[|\bar{X} - \mathbf{E}[\bar{X}]| > \varepsilon] \leq 2\exp\left(\frac{-2\varepsilon^2 n}{\Delta^2}\right).$$

Again this is a PAC bound, now with $\delta = 2\exp(-2\varepsilon^2 n/\Delta^2)$. For a desired error tolerance ε and failure probability δ, we can set $n = (\Delta^2/(2\varepsilon^2))\ln(2/\delta)$. Note that this has a similar relationship with ε as the Chebyshev bound, but the dependence of n on δ is exponentially less for this bound.

Relating this all back to the Gaussian distribution in the CLT, the Chebyshev bound only uses the variance information about the Gaussian, but the Chernoff-Hoeffding bound uses all of the "moments": that it decays exponentially.

These are the most basic and common PAC concentration of measure bounds, but are by no means exhaustive.

copyright: Jeff M. Phillips

Example: Uniform Distribution

Consider a random variable $X \sim f$ where $f(x) = \{\frac{1}{2}$ if $x \in [0,2]$ and 0 otherwise.$\}$, i.e, the Uniform distribution on $[0,2]$. We know $\mathbf{E}[X] = 1$ and $\mathbf{Var}[X] = \frac{1}{3}$.

- Using the Markov Inequality, we can say $\mathbf{Pr}[X > 1.5] \leq 1/(1.5) \approx 0.6666$ and $\mathbf{Pr}[X > 3] \leq 1/3 \approx 0.33333$.
  or $\mathbf{Pr}[X - \mu > 0.5] \leq \frac{2}{3}$ and $\mathbf{Pr}[X - \mu > 2] \leq \frac{1}{3}$.
- Using the Chebyshev Inequality, we can say that $\mathbf{Pr}[|X - \mu| > 0.5] \leq (1/3)/0.5^2 = \frac{4}{3}$ (which is meaningless). But $\mathbf{Pr}[|X - \mu| > 2] \leq (1/3)/(2^2) = \frac{1}{12} \approx 0.08333$.

Now consider a set of $n = 100$ random variables $X_1, X_2, \ldots, X_n$ all drawn iid from the same pdf f as above. Now we can examine the random variable $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. We know that $\mu_n = \mathbf{E}[\bar{X}] = \mu$ and that $\sigma_n^2 = \mathbf{Var}[\bar{X}] = \sigma^2/n = 1/(3n) = 1/300$.

- Using the Chebyshev Inequality, we can say that $\mathbf{Pr}[|\bar{X} - \mu| > 0.5] \leq \sigma_n^2/(0.5)^2 = \frac{1}{75} \approx 0.01333$, and $\mathbf{Pr}[|\bar{X} - \mu| > 2] \leq \sigma_n^2/2^2 = \frac{1}{1200} \approx 0.0008333$.
- Using the Chernoff-Hoeffding bound, we can say that $\mathbf{Pr}[|\bar{X} - \mu| > 0.5] \leq 2\exp(-2(0.5)^2 n/\Delta^2) = 2\exp(-100/8) \approx 0.0000074533$, and $\mathbf{Pr}[|\bar{X} - \mu| > 2] \leq 2\exp(-2(2)^2 n/\Delta^2) = 2\exp(-200) \approx 2.76 \cdot 10^{-87}$.

copyright: Jeff M. Phillips

Exercises

Q3.1: Consider a pdf f so that a random variable $X \sim f$ has expected value $\mathbf{E}[X] = 3$ and variance $\mathbf{Var}[X] = 10$. Now consider $n = 10$ iid random variables $X_1, X_2, \ldots, X_{10}$ drawn from f. Let $\bar{X} = \frac{1}{10}\sum_{i=1}^{10} X_i$.

1. What is $\mathbf{E}[\bar{X}]$?
2. What is $\mathbf{Var}[\bar{X}]$?
3. What is the standard deviation of $\bar{X}$?
4. Which is larger $\mathbf{Pr}[X > 4]$ or $\mathbf{Pr}[\bar{X} > 4]$?
5. Which is larger $\mathbf{Pr}[X > 2]$ or $\mathbf{Pr}[\bar{X} > 2]$?

Q3.2: Let X be a random variable that you know is in the range $[-1, 2]$ and you know has expected value of $\mathbf{E}[X] = 0$. Use the Markov Inequality to upper bound $\mathbf{Pr}[X > 1.5]$?
(Hint: you will need to use a change of variables.)

Q3.3: Consider a pdf f so that a random variable $X \sim f$ has expected value $\mathbf{E}[X] = 5$ and variance $\mathbf{Var}[X] = 100$. Now consider $n = 16$ iid random variables $X_1, X_2, \ldots, X_{16}$ drawn from f. Let $\bar{X} = \frac{1}{16}\sum_{i=1}^{16} X_i$.

1. What is $\mathbf{E}[\bar{X}]$?
2. What is $\mathbf{Var}[\bar{X}]$?

Assume we know that X is never smaller than 0 and never larger than 20.

3. Use the Markov inequality to upper bound $\mathbf{Pr}[\bar{X} > 8]$.
4. Use the Chebyshev inequality to upper bound $\mathbf{Pr}[\bar{X} > 8]$.
5. Use the Chernoff-Hoeffding inequality to upper bound $\mathbf{Pr}[\bar{X} > 8]$.
6. If we increase n to 100, how will the above three bounds be affected.

Q3.4: Consider a (parked) self-driving car that returns n iid estimates to the distance of a tree. We will model these n estimates as a set of n scalar random variables $X_1, X_2, \ldots, X_n$ taken iid from an unknown pdf f, which we assume models the true distance plus unbiased noise. (The sensor can take many iid estimates in rapid fire fashion.) The sensor is programmed to only return values between 0 and 20 feet, and that the variance of the sensing noise is 64 feet squared. Let $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. We want to understand as a function of n how close $\bar{X}$ is to μ, which is the true distance to the tree.

1. Use Chebyshev's Inequality to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 1] \leq 0.5$.
2. Use Chebyshev's Inequality to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 0.1] \leq 0.1$.
3. Use the Chernoff-Hoeffding bound to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 1] \leq 0.5$.
4. Use the Chernoff-Hoeffding bound to determine a value n so that $\mathbf{Pr}[|\bar{X} - \mu| \geq 0.1] \leq 0.1$.

Q3.5: Consider two random variables C and T describing how many coffees and teas I will buy in the coming week; clearly neither can be smaller than 0. Based on personal experience, I know the following summary statistics about my coffee and tea buying habits: $\mathbf{E}[C] = 3$ and $\mathbf{Var}[C] = 1$ also $\mathbf{E}[T] = 2$ and $\mathbf{Var}[T] = 5$.

1. Use Markov's Inequality to upper bound the probability I buy 4 or more coffees, and the same for teas: $\mathbf{Pr}[C \geq 4]$ and $\mathbf{Pr}[T \geq 4]$.

copyright: Jeff M. Phillips

2. Use Chebyshev's Inequality to upper bound the probability I buy 4 or more coffees, and the same for teas: $\mathbf{Pr}[C \geq 4]$ and $\mathbf{Pr}[T \geq 4]$.

Q3.6: The average score on a test is 82 with a standard deviation of 4 percentage points. All tests have scores between 0 and 100.

1. Using Chebyshev's inequality, at least what percentage of the tests have a grade of at least 70 and at most 94?
2. Using Markov's inequality, what is the highest percentage of tests which could have score less than 60?

 copyright: Jeff M. Phillips

4 Linear Algebra Review

For this topic we quickly review many key aspects of linear algebra that will be necessary for the remainder of the text.

4.1 Vectors and Matrices

For the context of data analysis, the critical part of linear algebra deals with vectors and matrices of real numbers.

In this context, a *vector* $v = (v_1, v_2, \ldots, v_d)$ is equivalent to a point in $\mathbb{R}^d$. By default a vector will be a column of d numbers (where d is context specific)

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

but in some cases we will assume the vector is a row

$$v^T = [v_1 \; v_2 \; \ldots \; v_n].$$

An $n \times d$ *matrix* A is then an ordered set of n row vectors $a_1, a_2, \ldots a_n$

$$A = [a_1; \; a_2; \; \ldots \; a_n] = \begin{bmatrix} - & a_1 & - \\ - & a_2 & - \\ & \vdots & \\ - & a_n & - \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & \ldots & A_{1,d} \\ A_{2,1} & A_{2,2} & \ldots & A_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \ldots & A_{n,d} \end{bmatrix},$$

where vector $a_i = [A_{i,1}, A_{i,2}, \ldots, A_{i,d}]$, and $A_{i,j}$ is the element of the matrix in the ith row and jth column. We can write $A \in \mathbb{R}^{n \times d}$ when it is defined on the reals.

A *transpose* operation $(\cdot)^T$ reverses the roles of the rows and columns, as seen above with vector v. For a matrix, we can write:

$$A^T = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \ldots & a_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{2,1} & \ldots & A_{n,1} \\ A_{1,2} & A_{2,2} & \ldots & A_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,d} & A_{2,d} & \ldots & A_{n,d} \end{bmatrix}.$$

Example: Linear Equations

A simple place these objects arise is in linear equations. For instance

$$\begin{array}{rrrl} 3x_1 & -7x_2 & +2x_3 & = -2 \\ -1x_1 & +2x_2 & -5x_3 & = 6 \end{array}$$

is a system of $n = 2$ linear equations, each with $d = 3$ variables. We can represent this system in matrix-vector notation as

$$Ax = b$$

where

$$b = \begin{bmatrix} -2 \\ 6 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} 3 & -7 & 2 \\ -1 & 2 & -5 \end{bmatrix}.$$

4.2 Addition

We can add together two vectors or two matrices only if they have the same dimensions. For vectors $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ and $y = (y_1, y_2, \ldots, y_d) \in \mathbb{R}^d$, then vector

$$z = x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_d + y_d) \in \mathbb{R}^d.$$

Similarly for two matrices $A, B \in \mathbb{R}^{n \times d}$, then $C = A + B$ is defined where $C_{i,j} = A_{i,j} + B_{i,j}$ for all i, j.

4.3 Multiplication

Multiplication only requires alignment along one dimension. For two matrices $A \in \mathbb{R}^{n \times d}$ and $B \in \mathbb{R}^{d \times m}$ we can obtain a new matrix $C = AB \in \mathbb{R}^{n \times m}$ where $C_{i,j}$, the element in the ith row and jth column of C is defined

$$C_{i,j} = \sum_{k=1}^{d} A_{i,k} B_{k,j}.$$

To multiply A times B (where A is to the left of B, the order matters!) then we require the row dimension d of A to match the column dimension d of B. If $n \neq m$, then we *cannot* multiply BA. Keep in mind:

- Matrix multiplication is *associative* $(AB)C = A(BC)$.
- Matrix multiplication is *distributive* $A(B + C) = AB + AC$.
- Matrix multiplication is ***not*** *commutative* $AB \neq BA$.

We can also multiply a matrix A by a scalar α. In this setting $\alpha A = A\alpha$ and is defined by a new matrix B where $B_{i,j} = \alpha A_{i,j}$.

vector-vector products. There are two types of vector-vector products, and their definitions follow directly from that of matrix-matrix multiplication (since a vector is a matrix where one of the dimensions is 1). But it is worth highlighting these.

Given two column vectors $x, y \in \mathbb{R}^d$, the *inner product* or *dot product* is written

$$x^T y = x \cdot y = \langle x, y \rangle = [x_1 \; x_2 \; \ldots \; x_d] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} = \sum_{i=1}^{d} x_i y_i,$$

copyright: Jeff M. Phillips

where x_i is the ith element of x and similar for y_i. This text will prefer the last notation $\langle x, y \rangle$ since the same can be used for row vectors, and there is no confusion with scalar multiplication in using $x \cdot y$. Whether a vector is a row or a column is often arbitrary, in a computer they are typically stored the same way in memory.

Note that this dot product operation produces a single scalar value. And it is a linear operator. So this means for any scalar value α and three vectors $x, y, z \in \mathbb{R}^d$ we have

$$\langle \alpha x, y + z \rangle = \alpha \langle x, y + z \rangle = \alpha \left(\langle x, y \rangle + \langle x, z \rangle \right).$$

This operation is associative, distributive, and commutative.

Geometry of the Dot Product

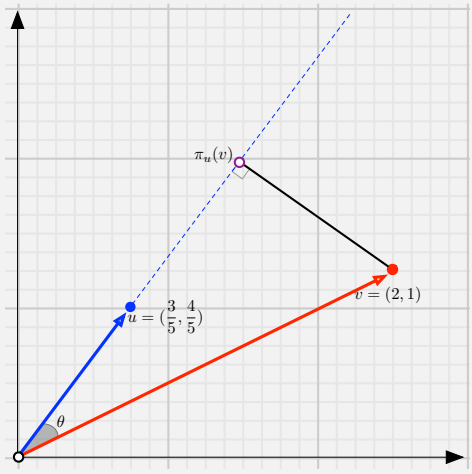
A dot product is one of my favorite mathematical operations! It encodes a lot of geometry. Consider two vectors $u = (\frac{3}{5}, \frac{4}{5})$ and $v = (2, 1)$, with an angle θ between them. Then it holds

$$\langle u, v \rangle = \text{length}(u) \cdot \text{length}(v) \cdot \cos(\theta).$$

Here $\text{length}(\cdot)$ measures the distance from the origin. We'll see how to measure length with a "norm" $\| \cdot \|$ soon.

Moreover, since $\|u\| = \text{length}(u) = 1$, then we can also interpret $\langle u, v \rangle$ as the length of v projected onto the line through u. That is, let $\pi_u(v)$ be the closest point to v on the line through u (the line through u and the line segment from v to $\pi_u(v)$ make a right angle). Then

$$\langle u, v \rangle = \text{length}(\pi_u(v)) = \|\pi_u(v)\|.$$



For two column vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^d$, the *outer product* is written

$$y^T x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \dots & y_d \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \dots & x_1 y_d \\ x_2 y_1 & x_2 y_2 & \dots & x_2 y_d \\ \vdots & \vdots & \ddots & \vdots \\ x_n y_1 & x_n y_2 & \dots & x_n y_d \end{bmatrix} \in \mathbb{R}^{n \times d}.$$

Note that the result here is a matrix, not a scalar. The dimensions are not required to match.

matrix-vector products. Another important and common operation is a matrix-vector product. Given a matrix $A \in \mathbb{R}^{n \times d}$ and a vector $x \in \mathbb{R}^d$, their product $y = Ax \in \mathbb{R}^n$.

 copyright: Jeff M. Phillips

When A is composed of row vectors $[a_1; a_2; \ldots; a_n]$, then it is useful to imagine this as transposing x (which should be a column vector here, so a row vector after transposing), and taking the dot product with each row of A. I like to think of this as x^T sliding down the rows of A, and for each row a_i outputting a scalar value $\langle a_i, x \rangle$ into the corresponding output vector.

$$y = Ax = \begin{bmatrix} - & a_1 & - \\ - & a_2 & - \\ & \vdots & \\ - & a_n & - \end{bmatrix} x = \begin{bmatrix} \langle a_1, x \rangle \\ \langle a_2, x \rangle \\ \vdots \\ \langle a_n, x \rangle \end{bmatrix}.$$

4.4 Norms

The standard *Euclidean norm* (think "length") of a vector $v = (v_1, v_2, \ldots, v_d) \in \mathbb{R}^d$ is defined

$$\|v\| = \sqrt{\sum_{i=1}^{d} v_i^2} = \sqrt{v_1 v_1 + v_2 v_2 + \ldots + v_d v_d} = \sqrt{\langle v, v \rangle}.$$

This measures the "straight-line" distance from the origin to the point at v. A vector v with norm $\|v\| = 1$ is said to be a *unit vector*; sometimes a vector x with $\|x\| = 1$ is said to be *normalized*.

However, a "norm" is a more generally concept. A class called L_p *norms* are well-defined for any parameter $p \in [1, \infty)$ as

$$\|v\|_p = \left(\sum_{i=1}^{d} |v_i|^p \right)^{1/p}.$$

Thus, when no p is specified, it is assumed to be $p = 2$. It is also common to denote $\|v\|_\infty = \max_{i=1}^{d} |v_i|$.

Because subtraction is well-defined between vectors $v, u \in \mathbb{R}^d$ of the same dimension, then we can also take the norm of $\|v - u\|_p$. While this is technically the norm of the vector resulting from the subtraction of u from v; it also provides a distance between u and v. In the case of $p = 2$, then

$$\|u - v\|_2 = \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2}$$

is precisely the straight-line (Euclidean) distance between u and v, and it is useful to think of them as points.

Moreover, all L_p norms define a distance $D_p(u, v) = \|u - v\|_p$, which satisfies a set of special properties required for a distance to be a *metric*. This include:

- **Symmetry:** For any $u, v \in \mathbb{R}^d$ we have $D(u, v) = D(v, u)$.
- **Non-negativity:** For any $u, v \in \mathbb{R}^d$ we have $D(u, v) \geq 0$, and $D(u, v) = 0$ if and only if $u = v$.
- **Triangle Inequality:** For any $u, v, w \in \mathbb{R}^d$ we have $D(u, w) + D(w, v) \geq D(u, v)$.

We can also define norms for matrices A. These take on slightly different notational conventions. The two most common are the spectral norm $\|A\| = \|A\|_2$ and the Frobenius norm $\|A\|_F$. The *Frobenius norm* is the most natural extension of the $p = 2$ norm for vectors, but uses a subscript F instead. It is defined for matrix $A \in \mathbb{R}^{n \times d}$ as

$$\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{d} A_{i,j}^2} = \sqrt{\sum_{i=1}^{n} \|a_i\|^2},$$

where $A_{i,j}$ is the element in the ith row and jth column of A, and where a_i is the ith row vector of A. The *spectral norm* is defined for a matrix $A \in \mathbb{R}^{n \times d}$ as

$$\|A\| = \|A\|_2 = \max_{\substack{x \in \mathbb{R}^d \\ \|x\| \neq 0}} \|Ax\|/\|x\| = \max_{\substack{y \in \mathbb{R}^n \\ \|y\| \neq 0}} \|yA\|/\|y\|.$$

Its useful to think of these x and y vectors as being unit vectors, then the denominator can be ignored (as they are 1). Then we see that x and y only contain "directional" information, and the $\arg\max$ vector (e.g., the x which maximizes $\|Ax\|/\|x\|$) point in the directions that maximize the norm.

4.5 Linear Independence

Consider a set of k vectors $x_1, x_2, \ldots, x_k \in \mathbb{R}^d$, and a set of k scalars $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{R}$. Then because of linearity of vectors, we can write a new vector in $\mathbb{R}^d$ as

$$z = \sum_{i=1}^{k} \alpha_i x_i.$$

For a set of vectors $X = \{x_1, x_2, \ldots, x_k\}$, for any vector z such that there exists a set of scalars α so z can be written as the above summation, then we say z is *linearly dependent* on X. If z **cannot** be written with any choice of α_is, the we say z is *linearly independent* of X. All vectors $z \in \mathbb{R}^d$ which are linearly dependent on X are said to be in its *span*.

$$\mathsf{span}(X) = \left\{ z \;\middle|\; z = \sum_{i=1}^{k} \alpha_i x_i, \;\; \alpha_i \in \mathbb{R} \right\}.$$

If $\mathsf{span}(X) = \mathbb{R}^d$ (that is for vectors $X = x_1, x_2, \ldots, x_k \in \mathbb{R}^d$ *all* vectors are in the span), then we say X forms a *basis*.

Example: Linear Independence

Consider input vectors in a set X as

$$x_1 = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} \qquad x_2 = \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix}$$

And two other vectors

$$z_1 = \begin{bmatrix} -3 \\ -5 \\ 2 \end{bmatrix} \qquad z_2 = \begin{bmatrix} 3 \\ 7 \\ 1 \end{bmatrix}$$

Note that z_1 is linearly dependent on X since it can be written as $z_1 = x_1 - 2x_2$ (here $\alpha_1 = 1$ and $\alpha_2 = -2$). However z_2 is linearly independent from X since there are no scalars α_1 and α_2 so that $z_2 = \alpha_1 x_1 + \alpha_2 x_2$ (we need $\alpha_1 = \alpha_2 = 1$ so the first two coordinates align, but then the third coordinate cannot).
Also the set X is linearly independent, since there is no way to write $x_2 = \alpha_1 x_1$.

A set of vectors $X = \{x_1, x_2, \ldots, x_n\}$ is *linearly independent* if there is no way to write any vector $x_i \in X$ in the set with scalars $\{\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n\}$ as the sum

$$x_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \alpha_j x_j$$

of the other vectors in the set.

4.6 Rank

The *rank* of a set of vectors $X = \{x_1, \ldots, x_n\}$ is the size of the largest subset $X' \subset X$ which are linearly independent. Usually we report $\mathsf{rank}(A)$ as the rank of a matrix A. It is defined as the rank of the rows of the matrix, or the rank of its columns; it turns out these quantities are always the same.

If $A \in \mathbb{R}^{n \times d}$, then $\mathsf{rank}(A) \leq \min\{n, d\}$. If $\mathsf{rank}(A) = \min\{n, d\}$, then A is said to be *full rank*. For instance, if $d < n$, then using the rows of $A = [a_1; a_2; \ldots; a_n]$, we can describe *any* vector $z \in \mathbb{R}^d$ as the linear combination of these rows: $z = \sum_{i=1}^{n} \alpha_i a_i$ for some set $\{\alpha_1, \ldots, \alpha_n\}$. In fact, if A is full rank we can do so and set all but d of these scalars to 0.

4.7 Inverse

A matrix A is said to be *square* if it has the same number of column as it has rows. A square matrix $A \in \mathbb{R}^{n \times n}$ may have an *inverse* denoted A^{-1}. If it exists, it is a unique matrix which satisfies:

$$A^{-1}A = I = AA^{-1}$$

where I is the $n \times n$ identity matrix

$$I = \begin{bmatrix} 1 & 0 & \ldots 0 & 0 & \\ 0 & 1 & \ldots 0 & 0 & \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix} = \mathsf{diag}(1, 1, \ldots, 1).$$

Note that I serves the purpose of 1 in scalar algebra, so for any (non-zero) scalar α then using $\alpha^{-1} = \frac{1}{\alpha}$ we have $\alpha\alpha^{-1} = 1 = \alpha^{-1}\alpha$.

A matrix is said to be *invertable* if it has an inverse. Only square, full-rank matrices are invertable; and a matrix is always invertable if it is square and full rank. If a matrix is not square, the inverse is not defined. If a matrix is not full rank, then it does not have an inverse.

4.8 Orthogonality

Two vectors $x, y \in \mathbb{R}^d$ are *orthogonal* if $\langle x, y \rangle = 0$. This means those vectors are at a right angle to each other.

Example: Orthongonality

Consider two vectors $x = (2, -3, 4, -1, 6)$ and $y = (4, 5, 3, -7, -2)$. They are orthogonal since

$$\langle x, y \rangle = (2 \cdot 4) + (-3 \cdot 5) + (4 \cdot 3) + (-1 \cdot -7) + (6 \cdot -2) = 8 - 15 + 12 + 7 - 12 = 0.$$

A square matrix $U \in \mathbb{R}^{n \times n}$ is *orthogonal* if all of its columns $[u_1, u_2, \ldots, u_n]$ are normalized and are all orthogonal with each other. It follows that

$$U^T U = I = UU^T$$

since for any normalized vector u that $\langle u, u \rangle = \|u\| = 1$, and any two distinct columns $u_i \neq u_j$ then $\langle u_i, u_j \rangle = 0$.

copyright: Jeff M. Phillips

A set of columns (for instance those of an orthogonal U) which are normalized and all orthogonal to each other are said to be *orthonormal*. If $U \in \mathbb{R}^{n \times d}$ and has orthonormal columns, then $U^T U = I$ (here I is $d \times d$) but $UU^T \neq I$.

Orthogonal matrices are norm preserving under multiplication. That means for an orthogonal matrix $U \in \mathbb{R}^{n \times n}$ and any vector $x \in \mathbb{R}^n$, then $\|Ux\| = \|x\|$.

Moreover, the columns $[u_1, u_2, \ldots, u_n]$ of an orthogonal matrix $U \in \mathbb{R}^{n \times n}$ form an *basis* for $\mathbb{R}^n$. This means that for any vector $x \in \mathbb{R}^n$, there exists a set of scalars $\alpha_1, \ldots, \alpha_n$ such that $x = \sum_{i=1}^n \alpha_i u_i$. More interestingly, we also have $\|x\|^2 = \sum_{i=1}^n \alpha_i^2$.

This can be interpreted as U describing a *rotation* (with possible mirror flips) to a new set of coordinates. That is the old coordinates of x are $(x_1, x_2, \ldots, x_n)$ and the coordinates in the new orthogonal basis $[u_1, u_2, \ldots, u_n]$ are $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

```
import numpy as np
from numpy import linalg as LA

#create an array, a row vector
v = np.array([1,2,7,5])
print v
#[1 2 7 5]
print v[2]
#7

#create a n=2 x d=3 matrix
A = np.array([[3,4,3],[1,6,7]])
print A
#[[3 4 3]
# [1 6 7]]
print A[1,2]
#7
print A[:, 1:3]
#[[4 3]
# [6 7]]

#adding and multiplying vectors
u = np.array([3,4,2,2])
#elementwise add
print v+u
#[4 6 9 7]
#elementwise multiply
print v*u
#[ 3  8  14 10]
# dot product
print v.dot(u)
# 35
print np.dot(u,v)
# 35

#matrix multiplication
B = np.array([[1,2],[6,5],[3,4]])
print A.dot(B)
#[[36 38]
# [58 60]]
x = np.array([3,4])
print B.dot(x)
#[11 38 25]

#norms
```

```
print LA.norm(v)
#8.88819441732
print LA.norm(v,1)
#15.0
print LA.norm(v,np.inf)
#7.0
print LA.norm(A, 'fro')
#10.9544511501
print LA.norm(A,2)
#10.704642743

#transpose
print A.T
#[[3 1]
# [4 6]
# [3 7]]
print x.T
#[3 4]   (always prints in row format)

print LA.matrix_rank(A)
#2
C = np.array([[1,2],[3,5]])
print LA.inv(C)
#[[-5.  2.]
# [ 3. -1.]]
print C.dot(LA.inv(C))
#[[  1.00000000e+00   2.22044605e-16]       (nearly  [[1 0]
# [  0.00000000e+00   1.00000000e+00]]                [0 1]] )
```

Exercises

Q4.1: Consider a matrix

$$A = \begin{bmatrix} 2 & 2 & 3 \\ -2 & 7 & 4 \\ -3 & -3 & -4 \\ -8 & 2 & 3 \end{bmatrix}.$$

1. Add a column to A so that it is invertable.
2. Remove a row from A so that it is invertable.
3. Is AA^T invertable?
4. Is A^TA invertable?

Q4.2: Consider two vectors $u = (0.5, 0.4, 0.4, 0.5, 0.1, 0.4, 0.1)$ and $v = (-1, -2, 1, -2, 3, 1, -5)$.

1. Check if u or v is a unit vector.
2. Calculate the dot product $\langle u, v \rangle$.
3. Are u and v orthogonal?

Q4.3: Consider the following 3 vectors in $\mathbb{R}^9$:

$$\begin{aligned} v &= (1, 2, 5, 2, -3, 1, 2, 6, 2) \\ u &= (-4, 3, -2, 2, 1, -3, 4, 1, -2) \\ w &= (3, 3, -3, -1, 6, -1, 2, -5, -7) \end{aligned}$$

Report the following:

1. $\langle v, w \rangle$
2. Are any pair of vectors orthogonal, and if so which ones?
3. $\|u\|_2$
4. $\|w\|_\infty$

Q4.4: Consider the following 3 matrices:

$$A = \begin{bmatrix} 2 & -2 \\ -3 & 1 \\ 5 & -3 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 4 & 4 \\ -2 & 3 & -7 \\ 2 & 5 & -7 \end{bmatrix} \quad C = \begin{bmatrix} 4 & -1 & 2 \\ -8 & 2 & -4 \\ 2 & 1 & -4 \end{bmatrix}$$

Report the following:

1. A^TB
2. $C + B$
3. Which matrices are full rank?
4. $\|C\|_F$
5. $\|A\|_2$
6. B^{-1}

copyright: Jeff M. Phillips

5 Linear Regression

We introduce the basic model of linear regression. It builds a linear model to predict one variable from one other variable or from a set of other variables. We will demonstrate how this simple technique can extend to building potentially much more complex polynomial models. Then we will introduce the central and extremely powerful idea of cross-validation. This method fundamentally changes the statistical goal of validating a model, to characterizing the data.

5.1 Simple Linear Regression

We will begin with the simplest form of linear regression. The input is a set of n 2-dimensional data points $(X, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. The ultimate goal will be to predict the y values using only the x-values. In this case x is the *explanatory variable* and y is the *dependent variable*.

The notation (X, y), with a uppercase X and lowercase y will become clear later since it will commonly generalize to multidimensional settings for the x-part, but stay 1-dimensional (or otherwise simple) for the y part.

In order to do this, we will "fit" a line through the data of the form

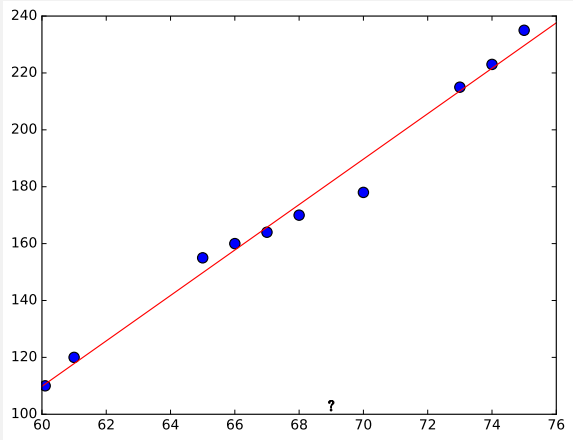
$$y = \ell(x) = ax + b,$$

where a (the slope) and b (the intercept) are parameters of this line. The line ℓ is our "model" for this input data.

Example: Fitting a line to height and weight

Consider the following data set that describes a set of heights and weights.

height (in)	weight (lbs)
66	160
68	170
60	110
70	178
65	155
61	120
74	223
73	215
75	235
67	164
69	?



Note that in the last entry, we have a height of 69, but we do not have a weight. If we were to guess the weight in the last row, how should we do this?

We can draw a line (the red one) through the data points. Then we can guess the weight for a data point with height 69, by the value of the line at height 69 inches: about 182 pounds.

Measuring error. The purpose of this line is not just to be close to all of the data (for this we will have to wait for PCA and dimensionality reduction). Rather, its goal is prediction; specifically, using the explanatory variable x to predict the dependent variable y.

In particular, for every value $x \in \mathbb{R}$, we can predict a value $\hat{y} = \ell(x)$. Then on our dataset, we can examine for each x_i how close $\hat{y}_i$ is to y_i. This difference is called a *residual*:

$$r_i = |y_i - \hat{y}_i| = |y_i - \ell(x_i)|.$$

Note that this residual is not the distance from y_i to the line ℓ, but the distance from y_i to the corresponding point with the same x-value. Again, this is because our only goal is prediction of y. And this will be important as it allows all of the techniques to be immune to the choice of units (e.g., inches or feet, pounds or kilograms)

So the residual measures the error of a single data point, but how should we measure the overall error of the entire data set? The common approach is the sum of squared errors:

$$\mathsf{SSE}((X, y), \ell) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \ell(x_i))^2.$$

Why is this the most common measure? Here are 3 explanations:

- The sum of squared errors was the optimal result for a single point estimator under Gaussian noise using Bayesian reasoning, when there was assumed Gaussian noise (See T2). In that case the answer was simply the mean of the data.
- If you treat the residuals as a vector $r = (r_1, r_2, \ldots, r_n)$, then the standard way to measure total length of a vector r is through its norm $\|r\|$, which is most commonly its 2-norm $\|r\| = \|r\|_2 = \sqrt{\sum_{i=1}^{n} r_i^2}$. The square root part is not so important (it does not change which line ℓ minimizes this error), so removing this square root, we are left with SSE.
- For this specific formulation, there is a simple closed form solution (which we will see next) for ℓ. And in fact, this solution will generalize to many more complex scenarios.

There are many other formulations of how best to measure error for the fit of a line (and other models), but we will not cover them in this class.

Solving for ℓ. To solve for the line which minimizes $\mathsf{SSE}((X, y), \ell)$ there is a very simply solution, in two steps. Calculate averages $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, and create centered n-dimension vectors $\bar{X} = (x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_n - \bar{x})$ for all x-coordinates and $\bar{Y} = (y_1 - \bar{y}, y_2 - \bar{y}, \ldots, y_n - \bar{y})$ for all y-coordinates.

1. Set $a = \langle \bar{Y}, \bar{X} \rangle / \|\bar{X}\|^2$
2. Set $b = \bar{y} - a\bar{x}$

This defines $\ell(x) = ax + b$.

We will provide the proof for why this is the optimal solution for the high-dimensional case (in short, it can be shown by expanding out the SSE expression, taking the derivative, and solving for 0). We will only provide some intuition here.

First lets examine the intercept

$$b = \frac{1}{n}\sum_{i=1}^{n} (y_i - ax_i) = \bar{y} - a\bar{x}$$

This setting of b ensures that the line $y = \ell(x) = ax + b$ goes through the point $(\bar{x}, \bar{y})$ at the center of the data set since $\bar{y} = \ell(\bar{x}) = a\bar{x} + b$.

Second, to understand how the slope a is chosen, it is illustrative to reexamine the dot product as

$$a = \frac{\langle \bar{Y}, \bar{X} \rangle}{\|\bar{X}\|^2} = \frac{\|\bar{Y}\| \cdot \|\bar{X}\| \cdot \cos\theta}{\|\bar{X}\|^2} = \frac{\|\bar{Y}\|}{\|\bar{X}\|} \cos\theta,$$

where θ is the angle between the n-dimensional vectors $\bar{Y}$ and $\bar{X}$. Now in this expression, the $\|\bar{Y}\|/\|\bar{X}\|$ captures how much on (root-squared) average $\bar{Y}$ increases as $\bar{X}$ does (the rise-over-run interpretation of slope). However, we may want this to be negative if there is a negative correlation between $\bar{X}$ and $\bar{Y}$, or really this does not matter much if there is no correlation. So the $\cos\theta$ term captures the correlation after normalizing the units of $\bar{X}$ and $\bar{Y}$.

```
import numpy as np

x = np.array([66, 68, 65, 70, 65, 62, 74, 70, 71, 67])
y = np.array([160, 170, 159, 188, 150, 120, 233, 198, 201, 164])

ave_x = np.average(x)
ave_y = np.average(y)

#first center the data points
xc = x - ave_x
yc = y - ave_y

a = xc.dot(yc)/xc.dot(xc)
b = ave_y - a*ave_x
print a, b

#or with scipy
from scipy import polyfit
(a,b)=polyfit(x,y,1)
print a, b

#predict weight at x=69
w=a*69+b
```

5.2 Linear Regression with Multiple Explanatory Variables

Magically, using linear algebra, everything extends gracefully to using more than one explanatory variables. Now consider a data set $(X, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where each data point has $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,d}) \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. That is there are d explanatory variables, as the coordinates of x_i, and one dependent variable in y_i. We would now like to use all of these variables at once to make a single (linear) prediction about the variable y_i. That is, we would like to create a model

$$\begin{aligned}\hat{y} = M_\alpha(x_i) = M_\alpha(x_{i,1}, x_{i,2}, \ldots, x_{i,d}) &= \alpha_0 + \sum_{j=1}^{d} \alpha_j x_{i,j} \\ &= \alpha_0 + \alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \ldots + \alpha_d x_{i,d}. \\ &= \langle \alpha, (1, x_{i,1}, x_{i,2}, \ldots, x_{i,d}) \rangle = \langle \alpha, (1, x_i) \rangle.\end{aligned}$$

In the above equivalent notations α_0 serves the purpose of the intercept b, and all of the α_is replace the single coefficient a in the simple linear regression. Indeed, we can write this model as a dot product between the $(d+1)$-dimensional vectors $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ and $(1, x_{i,1}, x_{i,2}, \ldots, x_{i,d}) = (1, x_i)$. As promised, the magic of linear algebra has allowed us to describe a more complex linear model M_α. Next we will see how to solve it.

Example: Predicting customer value

A website specializing in dongles (dongles-r-us.com) wants to predict the total dollar amount that visitors will spend on their site. It has installed some software that can track three variables:

- time (the amount of time on the page in seconds): X_1,
- jiggle (the amount of mouse movement in cm): X_2, and
- scroll (how far they scroll the page down in cm): X_3.

Also, for a set of past customers they have recorded the

- sales (how much they spend on dongles in cents): y.

We see a portion of their data set here with $n = 11$ customers:

time: X_1	jiggle: X_2	scroll: X_3	sales: y
232	33	402	2201
10	22	160	0
6437	343	231	7650
512	101	17	5599
441	212	55	8900
453	53	99	1742
2	2	10	0
332	79	154	1215
182	20	89	699
123	223	12	2101
424	32	15	8789

To build a model, we recast the data as an 11×4 matrix $X = [\mathbf{1}, X_1, X_2, X_3]$. We let y be the 11-dimensional column vector.

$$X = \begin{bmatrix} 1 & 232 & 33 & 402 \\ 1 & 10 & 22 & 160 \\ 1 & 6437 & 343 & 231 \\ 1 & 512 & 101 & 17 \\ 1 & 441 & 212 & 55 \\ 1 & 453 & 53 & 99 \\ 1 & 2 & 2 & 10 \\ 1 & 332 & 79 & 154 \\ 1 & 182 & 20 & 89 \\ 1 & 123 & 223 & 12 \\ 1 & 424 & 32 & 15 \end{bmatrix} \quad y = \begin{bmatrix} 2201 \\ 0 \\ 7650 \\ 5599 \\ 8900 \\ 1742 \\ 0 \\ 1215 \\ 699 \\ 2101 \\ 8789 \end{bmatrix}$$

The goal is to learn the 4-dimensional column vector $\alpha = [\alpha_0; \alpha_1; \alpha_2, \alpha_3]$ so

$$y \approx X\alpha.$$

Setting $\alpha = (X^T X)^{-1} X^T y$ obtains (roughly) $\alpha_0 = 2626$, $\alpha_1 = 0.42$, $\alpha_2 = 12.72$, and $\alpha_3 = -6.50$. This implies an average customer with no interaction on the site generates $\alpha_0 = \$2.62$. That time does not have a strong effect here (only a coefficient α_1 at only 0.42), but jiggle has a strong correlation (with coefficient $\alpha_2 = 12.72$, this indicates 12 cents for every centimeter of mouse movement). Meanwhile scroll has a negative effect (with coefficient $\alpha_3 = -6.5$); this means that the more they scroll, the less likely they are to spend (just browsing dongles!).

copyright: Jeff M. Phillips

Given a data point $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,d})$, we can again evaluate our prediction $\hat{y}_i = M(x_i)$ using the residual value $r_i = |y_i - \hat{y}_i| = |y_i = M(x_i)|$. And to evaluate a set of n data points, it is standard to consider the sum of squared error as

$$\mathsf{SSE}(X, y, M) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - M(x_i))^2.$$

To obtain the coefficients which minimize this error, we can now do so with very simple linear algebra.

First we construct a $n \times (d+1)$ data matrix $\tilde{X} = [\mathbf{1}, X_1, X_2, \ldots, X_d]$, where the first column $\mathbf{1}$ is the n-dimensional all ones column vector $[1; 1; \ldots; 1]$. Each of the next d columns is a column vector X_j, where $x_{i,j} = X_{i,j}$ is the ith entry of X_j and represents the jth coordinate of data point x_i. Then we let y be a n-dimensional column vector containing all the dependent variables. Now we can simply calculate the $(d+1)$-dimensional column vector $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ as

$$\alpha = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T y.$$

Let us compare to the simple case where we have 1 explanatory variable. The $(\tilde{X}^T \tilde{X})^{-1}$ term replaces the $\frac{1}{\|X\|^2}$ term. The $\tilde{X}^T y$ replaces the dot product $\langle \bar{Y}, \bar{X} \rangle$. And we do not need to separately solve for the intercept b, since we have created a new column in $\tilde{X}$ of all 1s. For any dependent data values, we multiply the found coefficient α_0 by an imaginary 1 data value.

Often the matrices X and $\tilde{X}$ are used interchangeably, and hence we drop the $\tilde{}$ from $\tilde{X}$ in most situations. We can either simply treat all data points x_i as one-dimension larger (with always a 1 in the first coordinate), or we can fit a model on the original matrix X and ignore the offset parameter α_0, which is then by default 0. The former approach, where each x_i is just assumed one dimension larger is more common since it automatically handles the offset parameter.

 copyright: Jeff M. Phillips

Geometry of the Normal Equations

Why does $\alpha = (X^T X)^{-1} X^T y$ minimize the sum of squared errors:

$$\mathsf{SSE}(X, y, M) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \langle \alpha, x_i \rangle)^2?$$

Fixing the data X and y, and representing M by its parameters α, we can consider a function $S(\alpha) = \mathsf{SSE}(X, y, M)$. Then we observe that $S(\alpha)$ is a quadratic function in α, and it is convex (see T6), so its minimum is when the gradient is 0. This will be true when each partial derivative is 0. Let $X_{i,j}$ be the jth coordinate of x_i so residual $r_i = (y_i - \langle \alpha, x_i \rangle)$ has partial derivative $\frac{\mathrm{d}r_i}{\mathrm{d}\alpha_j} = -X_{i,j}$. Then set to 0 each partial derivative:

$$0 = \frac{\mathrm{d}S(\alpha)}{\mathrm{d}\alpha_j} = 2 \sum_{i=1}^{n} r_i \frac{\mathrm{d}r_i}{\mathrm{d}\alpha_j} = 2 \sum_{i=1}^{n} r_i (-X_{i,j}) = 2 \sum_{i=1}^{n} (y_i - \langle \alpha, x_i \rangle)(-X_{i,j})$$

We can rearrange this into the *normal equations*

$$\sum_{i=1}^{n} X_{i,j} \langle x_i, \alpha \rangle = \sum_{i=1}^{n} X_{i,j} y_i \qquad \text{for all } j \in \{1, 2, \ldots, d\}$$

$$\text{equivalently} \quad (X^T X)\alpha = X^T y.$$

Multiplying by the inverse *gram matrix* $X^T X$ on both sides reveals the desired $\alpha = (X^T X)^{-1} X^T y$.

Geometry: To see why these are called the normal equations, consider a form

$$\mathbf{0} = (y - X\alpha)^T X,$$

where $\mathbf{0}$ is the all-zeros vector in $\mathbb{R}^n$. Thus for any vector $v \in \mathbb{R}^n$, then

$$0 = (y - X\alpha)^T X v = \langle y - X\alpha, Xv \rangle.$$

This includes when $v = \alpha$; under this setting then $Xv = X\alpha = \hat{y}$. Notice that $r = y - X\alpha$ is the vector in $\mathbb{R}^n$ that stores all residuals (so $y = \hat{y} + r$). Then the normal equations implies that

$$0 = \langle y - X\alpha, X\alpha \rangle = \langle r, \hat{y} \rangle;$$

that is, for the optimal α, the prediction $\hat{y}$ and the residual vector r are orthogonal. Since $\hat{y} = X\alpha$ is restricted to the $(d+1)$-dimensional span of the columns of X, and α minimizes $\|r\|^2$, then this orthogonality implies that r is the *normal* vector to this $(d+1)$-dimensional subspace.

```python
import numpy as np
from numpy import linalg as LA

# directly
alpha = np.dot(np.dot(LA.inv(np.dot(X.T,X)),X.T),y.T)

# or with LA.lstsq
alpha = LA.lstsq(X,y)[0]
```


copyright: Jeff M. Phillips

5.3 Polynomial Regression

Sometimes linear relations are not sufficient to capture the true pattern going on in the data with even a single dependent variable x. Instead we would like to build a model of the form:

$$\hat{y} = M_2(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2$$

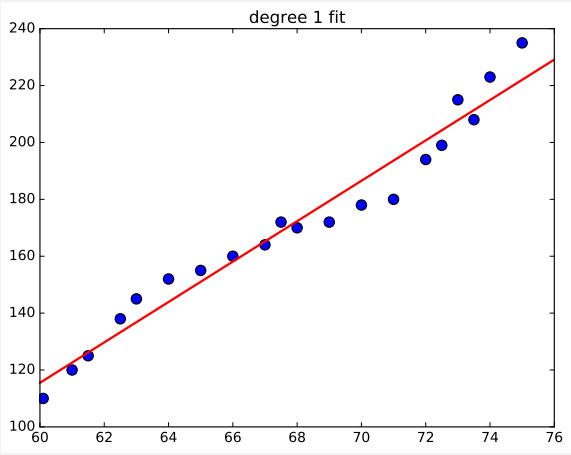
or more generally for some polynomial of degree p

$$\begin{aligned}\hat{y} = M_p(x) &= \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_p x^p \\ &= \alpha_0 + \sum_{i=1}^{p} \alpha_i x^i.\end{aligned}$$

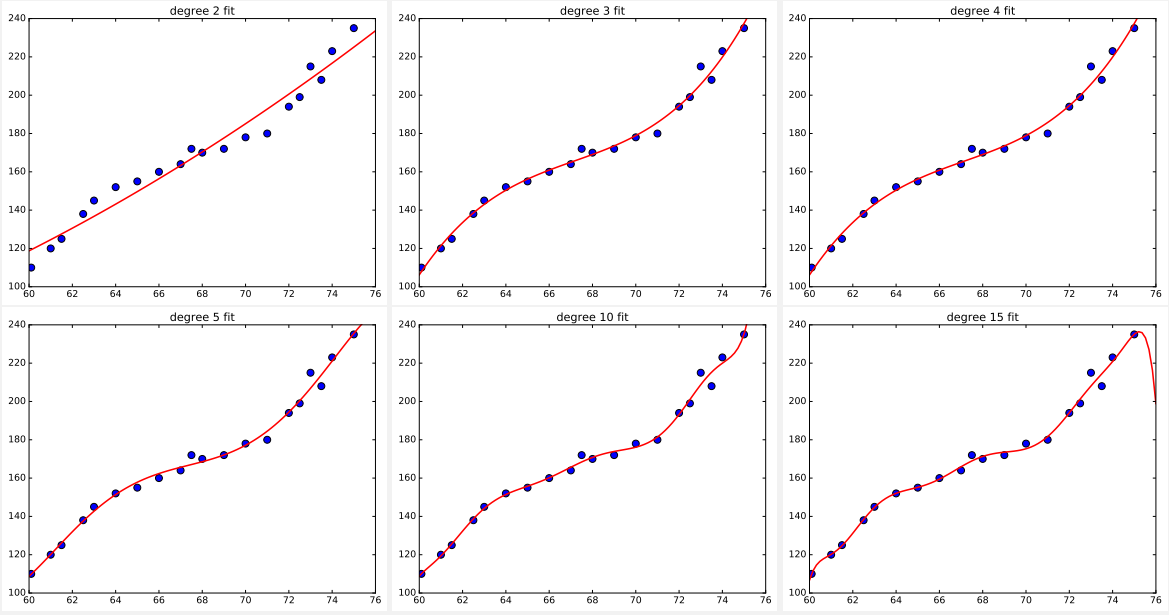
Example: Predicting Height and Weight with Polynomials

We found more height and weight data, in addition to the ones in the height-weight example above.

height (in)	weight (lbs)
61.5	125
73.5	208
62.5	138
63	145
64	152
71	180
69	172
72.5	199
72	194
67.5	172



But can we do better if we fit with a polynomial?



 copyright: Jeff M. Phillips

Again we can measure error for a single data point (x_i, y_i) as a residual as $r_i = |\hat{y} - y_i| = |M_\alpha(x_i) - y_i|$ and the error on n data points as the sum of squared residuals

$$\mathsf{SSE}(P, M_\alpha) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (M_p(x_i) - y_i)^2.$$

Under this error measure, it turns out we can again find a simple solution for the residuals $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_p)$. For each dependent variable data value x we create a $(p+1)$-dimensional vector

$$v = (1, x, x^2, \ldots, x^p).$$

And then for n data points $(x_1, y_1), \ldots, (x_n, y_n)$ we can create an $n \times (p+1)$ data matrix

$$\tilde{X}_p = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^p \\ 1 & x_2 & x_2^2 & \ldots & x_2^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^p \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ v_n \end{bmatrix}.$$

Then we can solve the same way as if each data value raised to a different power was a different dependent variable. That is we can solve for the coefficients $\alpha = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_p)$ as

$$\alpha = (\tilde{X}_p^T \tilde{X}_p)^{-1} \tilde{X}_p^T y.$$

5.4 Cross Validation

So how do we choose the correct value of p, the degree of the polynomial fit?

A (very basic) statistical (hypothesis testing) approach may be choose a model of the data (the best fit curve for some polynomial degree p, and assume Gaussian noise), then calculate the probability that the data fell outside the error bounds of that model. But maybe many different polynomials are a good fit?

In fact, if we choose p as $n - 1$ or greater, then the curve will *polynomially interpolate* all of the points. That is, it will pass through all points, so all points have a residual of exactly 0 (up to numerical precision). This is the basis of a lot of geometric modeling (e.g., for CAD), but bad for data modeling.

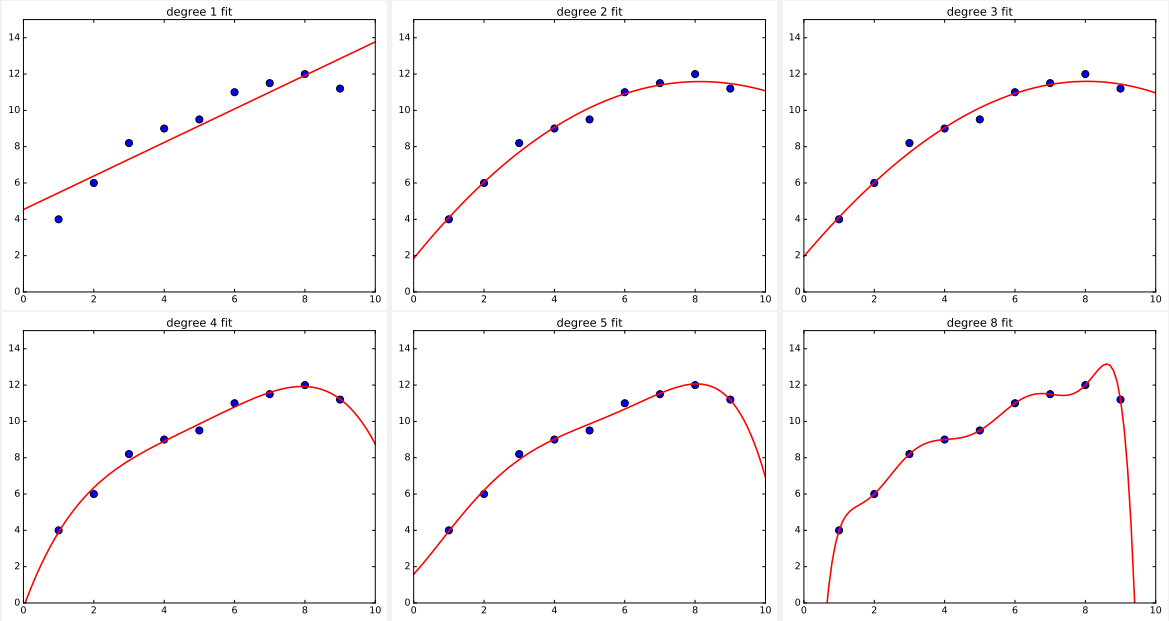
copyright: Jeff M. Phillips

Example: Simple polynomial example

Consider the simple data set of 9 points

x	1	2	3	4	5	6	7	8	9
y	4	6	8.2	9	9.5	11	11.5	12	11.2

With the following polynomial fits for $p = \{1, 2, 3, 4, 5, 8\}$. Believe your eyes, for $p = 8$, the curve actually passes through each and every point exactly.



Recall, our goal was for a new data point with only an x value to predict its y-value. Which do you think does the best job?

Generalization and Cross-Validation. Our ultimate goal in regression is *generalization* (how well do we do on new data), not SSE! Using some error measure (SSE) to fit a line or curve, is a good proxy for what we want, but in many cases (as with polynomial regression), it can be abused. We want to know how our model will generalize to new data. How would we measure this without new data?

The solution is *cross-validation*. In the simplest form, we **randomly** split our data into training data (on which we build a model) and testing data (on which we evaluate our model). The testing serves to estimate how well we would do on future data which we do not have.

- *Why randomly?:* Because you do not want to bias the model to do better on some parts than other in how you choose the split. Also, since we assume the data elements come iid from some underlying distribution, then the test data is also iid if you chose it randomly.

- *How large should the test data be?:* It depends on the data set. Both 10% and 33% are common.

Let (X, y) be the full data set (with n rows of data), and we split it into data sets $(X_{\text{train}}, y_{\text{train}})$ and $(X_{\text{test}}, y_{\text{test}})$ with n_{train} and n_{test} rows, respectively. With $n = n_{\text{train}} + n_{\text{test}}$. Next we build a model with the training data, e.g.,

$$\alpha = (X_{\text{train}}^T X_{\text{train}})^{-1} X_{\text{train}}^T y_{\text{train}}.$$

 copyright: Jeff M. Phillips

Then we evaluate the model M_α on the test data X_{test}, often using $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha)$ as

$$\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha) = \sum_{(x_i,y_i)\in(X_{\text{test}},y_{\text{test}})} (y_i - M_\alpha(x_i))^2 = \sum_{(x_i,y_i)\in(X_{\text{test}},y_{\text{test}})} (y_i - \langle (x_i;1), \alpha\rangle)^2.$$

We can use the testing data for two purposes:

- To estimate how well our model would perform on new data, yet unseen. That is the predicted residual of a new data point is precisely $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha)/n_{\text{test}}$.
- To choose the correct parameter for a model (which p to use)?

Its important to not use the same $(X_{\text{test}}, y_{\text{test}})$ to do both tasks. If we choose a model with $(X_{\text{test}}, y_{\text{test}})$, then we should reserve even more data for predicting the generalization error. When using the test data to choose a model parameter, then it is being used to build the model; thus evaluating generalization with this same data can suffer the same fate as testing and training with the same data.

So how should we choose the best p? We calculate models M_{α_p} for each value p on the same training data. Then calculate the model error $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_{\alpha_p})$ for each p, and see which has the smallest value. That is we train on $(X_{\text{train}}, y_{\text{train}})$ and test(evaluate) on $(X_{\text{test}}, y_{\text{test}})$.

Example: Simple polynomial example with Cross Validation

Now split our data sets into a train set and a test set:

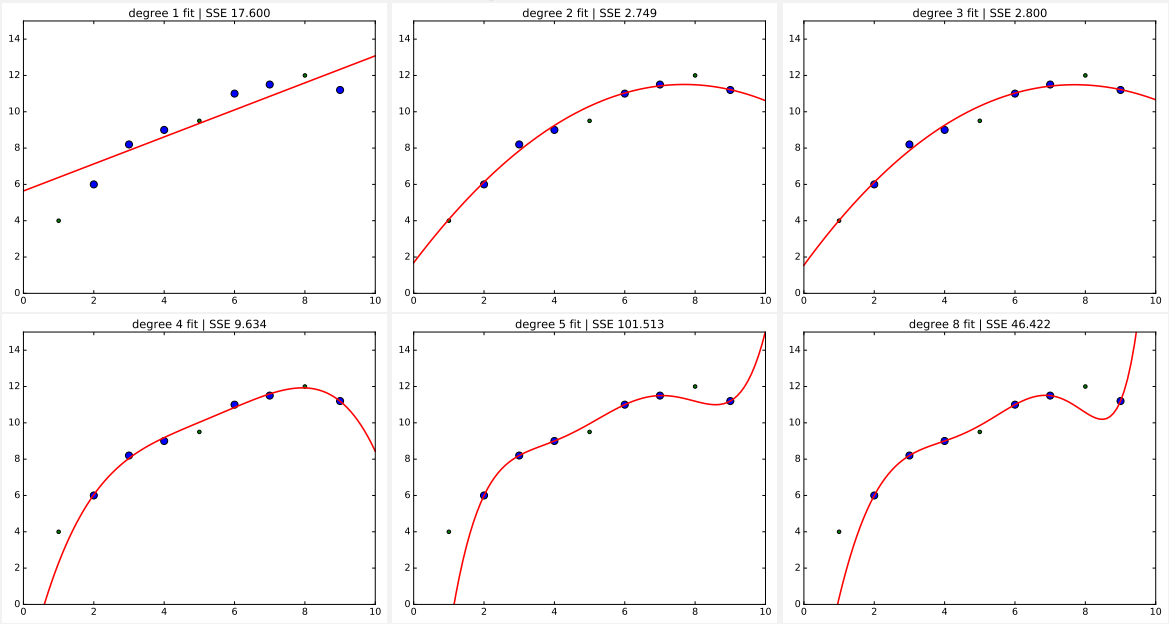
train:

x	2	3	4	6	7	8
y	6	8.2	9	11	11.5	12

test:

x	1	5	9
y	4	9.5	11.2

With the following polynomial fits for $p = \{1, 2, 3, 4, 5, 8\}$ generating model M_{α_p} on the test data. We then calculate the $\mathsf{SSE}(x_{\text{test}}, y_{\text{test}}, M_{\alpha_p})$ score for each (as shown):



And the polynomial model with degree $p = 2$ has the lowest SSE score of 2.749. It is also the simplest model that does a very good job by the "eye-ball" test. So we would choose this as our model.

copyright: Jeff M. Phillips

Leave-one-out Cross Validation. But, not training on the test data means that you use less data, and your model is worse! If your data is limited, you may not want to "waste" this data!

If your data is very large, then leaving out 10% is not a big deal. But if you only have 9 data points it can be. The smallest the test set could be is 1 point. But then it is not a very good representation of the full data set.

The alternative is to create n different training sets, each of size $n-1$ ($X_{1,\text{train}}, X_{2,\text{train}}, \ldots, X_{n,\text{train}}$), where $X_{i,\text{train}}$ contains all points except for x_i, which is a one-point test set. Then we build n different models $M_1, M_2, \ldots, M_n$, evaluate each model M_i on the one test point x_i to get an error $E_i = (y_i - M_i(x_i))^2$, and average their errors $E = \frac{1}{n}\sum_{i=1}^{n} E_i$. Again, the parameter with the smallest associated average error E is deemed the best. This allows you to build a model on as much data as possible, while still using all of the data to test.

However, this requires roughly n times as long to compute as the other techniques, so is often too slow for really big data sets.

 copyright: Jeff M. Phillips

```
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt
import scipy as sp
import numpy as np
import math
from numpy import linalg as LA

def plot_poly(x,y,xE,yE,p):
  plt.scatter(x,y, s=80, c="blue")
  plt.scatter(xE,yE, s=20, c="green")
  plt.axis([0,10,0,15])

  s=sp.linspace(0,10,101)

  coefs=sp.polyfit(x,y,p)
  ffit = np.poly1d(coefs)
  plt.plot(s,ffit(s),'r-',linewidth=2.0)

  #evaluate on xE, yE
  resid = ffit(xE)
  RSSE = LA.norm(resid-yE)
  SSE = RMSE * RMSE

  title = "degree␣%s␣fit␣|␣SSE␣%0.3f" % (p, SSE)
  plt.title(title)
  file = "CVpolyReg%s.pdf" % p
  plt.savefig(file, bbox_inches='tight')
  plt.clf()
  plt.cla()



# train data
xT = np.array([2, 3,   4,  6,  7,     9])
yT = np.array([6, 8.2, 9,  11, 11.5,  11.2])

#test data
xE = np.array([1, 5,   8])
yE = np.array([4, 9.5, 12])


p_vals = [1,2,3,4,5,8]
for i in p_vals:
  plot_poly(xT,yT,xE,yE,i)
```


copyright: Jeff M. Phillips

Exercises

We will use a dataset found here:
`http://www.cs.utah.edu/~jeffp/teaching/FoDA/D3.csv`

Q5.1: Let the first column of the data set be the explanatory variable `x`, and let the fourth column be the dependent variable `y`. [That is: ignore columns 2 and 3 for now]

1. Run simple linear regression to predict `y` from `x`. Report the linear model you found. Predict the value of `y` for new `x` values 0.3, for 0.5, and for 0.8.
2. Use cross-validation to predict generalization error, with error of a single data point (`x`, `y`) from a model M as $(M(\texttt{x}) - \texttt{y})^2$. Describe how you did this, and which data was used for what.
3. On the same data, run polynomial regression for $p = 2, 3, 4, 5$. Report polynomial models for each. With each of these models, predict the value of `y` for a new `x` values of 0.3, for 0.5, and for 0.8.
4. Cross-validate to choose the best model. Describe how you did this, and which data was used for what.

Q5.2: Now let the first three columns of the data set be separate explanatory variables `x1`, `x2`, `x3`. Again let the fourth column be the dependent variable `y`.

- Run linear regression simultaneously using all three explanatory variables. Report the linear model you found. Predict the value of `y` for new (`x1`,`x2`,`x3`) values $(0.3, 0.4, 0.1)$, for $(0.5, 0.2, 0.4)$, and for $(0.8, 0.2, 0.7)$.
- Use cross-validation to predict generalization error; as usual define the error of a single data point (`x1`, `x2`, `x3`, `y`) from a model M as $(M(\texttt{x1}, \texttt{x2}, \texttt{x3}) - \texttt{y})^2$. Describe how you did this, and which data was used for what.

Q5.3: Consider a data set (X, y) where $X \in \mathbb{R}^{n \times 3}$; and its decomposition into a test $(X_{\mathsf{test}}, y_{\mathsf{test}})$ and a training data set $(X_{\mathsf{train}}, y_{\mathsf{train}})$. Assume that X_{train} is not just a subset of X, but also prepends a columns of all 1s. We build a linear model

$$\alpha = (X_{\mathsf{train}}^T X_{\mathsf{train}})^{-1} X_{\mathsf{train}}^T y_{\mathsf{train}}.$$

where $\alpha \in \mathbb{R}^4$. The test data $(X_{\mathsf{test}}, y_{\mathsf{test}})$ consists of two data points: (x_1, y_1) and (x_2, y_2), where $x_1, x_2 \in \mathbb{R}^3$. Explain how to use (write a mathematical expression) this test data to estimate the generalization error. That is, if one new data point arrives x, how much squared error would we expect the model α to have compared to the unknown true value y?

 copyright: Jeff M. Phillips

6 Gradient Descent

In this topic we will discuss optimizing over general functions f. Typically the function is defined $f : \mathbb{R}^d \to \mathbb{R}$; that is its domain is multi-dimensional (in this case d-dimensional point α) and output is a real scalar ($\mathbb{R}$). This often arises to describe the "cost" of a model which has d parameters which describe the model (e.g., degree $(d-1)$-polynomial regression) and the goal is to find the parameters $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ with minimum cost. Although there are special cases where we can solve for these optimal parameters exactly, there are many cases where we cannot. What remains in these cases is to analyze the function f, and try to find its minimum point. The most common solution for this is gradient descent where we try to "walk" in a direction so the function decreases until we no longer can.

6.1 Functions

We review some basic properties of a function $f : \mathbb{R}^d \to \mathbb{R}$. Again, the goal will be to unveil abstract tools that are often easy to imagine in low dimensions, but automatically generalize to high-dimensional data. We will first provide definitions without any calculous.

Let $B_r(\alpha)$ define a Euclidean ball around a point $\alpha \in \mathbb{R}^d$ of radius r. That is, it includes all points $\{p \in \mathbb{R}^d \mid \|\alpha - p\| \leq r\}$, within a Euclidean distance of r from α. We will use $B_r(\alpha)$ to define a *local neighborhood* around a point α. The idea of "local" is quite flexible, and we can use *any* value of $r > 0$, basically it can be as small as we need it to be, as long as it is strictly greater than 0.

Minima and maxima. A *local maximum* of f is a point $\alpha \in \mathbb{R}^d$ so for some neighborhood $B_r(\alpha)$, all points $p \in B_r(\alpha)$ have smaller (or equal) function value than at α: $f(p) \leq f(\alpha)$. A *local minimum* of f is a point $\alpha \in \mathbb{R}^d$ so for some neighborhood $B_r(\alpha)$, all points $p \in B_r(\alpha)$ have larger (or equal) function value than at α: $f(p) \geq f(\alpha)$. If we remove the "or equal" condition for both definitions for $p \in B_r(\alpha), p \neq \alpha$, we say the maximum or minimum points are *strict*.

A point $\alpha \in \mathbb{R}^d$ is a *global maximum* of f if for all $p \in \mathbb{R}^d$, then $f(p) \leq f(\alpha)$. Likewise, a point $\alpha \in \mathbb{R}^d$ is a *global minimum* if for all $p \in \mathbb{R}^d$, then $f(p) \geq f(\alpha)$. There may be multiple global minimum and maximum. If there is exactly one point $\alpha \in \mathbb{R}^d$ that is a global minimum or global maximum, we again say it is *strict*.

When we just use the term *minimum* or *maximum* (without local or global) it implies a local minimum or maximum.

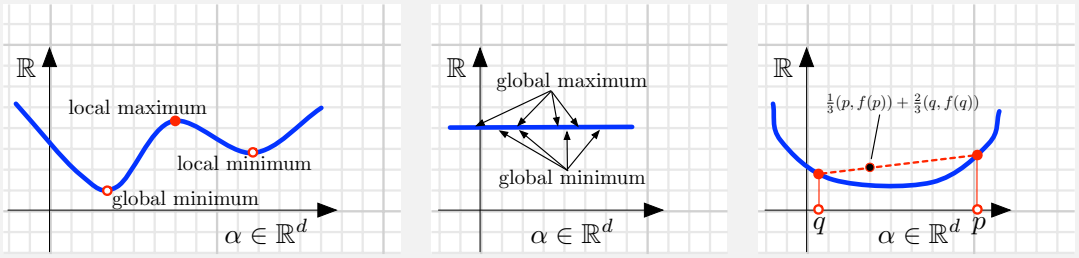
Focusing on a function restricted to a closed and bounded subset $S \subset \mathbb{R}^d$, if the function is continuous (that is, there exists a δ such that for all $\alpha \in \mathbb{R}^d$, there exists a radius r_δ such that if $p \in B_{r_\delta}(\alpha)$, then $|f(\alpha) - f(p)| \leq \delta$), then the function must have a global minimum and a global maximum. It may occur on the boundary of S.

A *saddle* point $\alpha \in \mathbb{R}^d$ (for $d > 1$) has within any neighborhood $B_r(\alpha)$ a points $p \in B_r(\alpha)$ with $f(p) < f(\alpha)$ (the lower points) and $p' \in B_r(\alpha)$ with $f(p') > f(\alpha)$ (the upper points). In particular, it is a saddle if within $B_r(\alpha)$ there are disconnected regions of upper points (and of lower points). The notion of saddle point is defined differently for $d = 1$. If these regions are connected, and it is not a minimum or maximum, then it is a *regular point*.

For an arbitrary (or randomly) chosen point α, it is usually a regular point (except for examples you are unlikely to encounter, the set of minimum, maximum, and saddle points are finite, while the set of regular points is infinite).

Example: Continuous Functions

Here we show some example functions, where the α-value on the x-axis represents a d-dimensional space. The first function has local minimum and maximum. The second function has a constant value, so every point is a global minimum and a global maximum. The third function f is convex, which is demonstrated with the line segment between points $(p, f(p))$ and $(q, f(q))$ is always above the function f.



Convex functions. In many cases we will assume (or at least desire) that our function is convex.

To define this it will be useful to define a line $\ell \subset \mathbb{R}^d$ as follows with any two points $p, q \in \mathbb{R}^d$. Then a line $\ell_{p,q}$ is the set of all points defined by any scalar $\lambda \in \mathbb{R}$ as

$$\ell_{p,q} = \{x = \lambda p + (1-\lambda)q \mid \lambda \in \mathbb{R}\}.$$

When $\lambda \in [0, 1]$, then this defines the line segment between p and q.

A function is *convex* if for any two points $p, q \in \mathbb{R}^d$, on the line segment between them has value less than (or equal) to the values at the weighted average of p and q. That is, it is convex if

$$\text{For all } p, q \in \mathbb{R} \quad \text{and for all } \lambda \in [0, 1] \quad f(\lambda p + (1-\lambda)q) \leq \lambda f(p) + (1-\lambda) f(q).$$

Removing the "or equal" condition, the function becomes *strictly convex*.

There are many very cool properties of convex functions. For instance, for two convex functions f and g, then $h(\alpha) = f(\alpha) + g(\alpha)$ is convex and so is $h(\alpha) = \max\{f(\alpha), g(\alpha)\}$. But one will be most important for us:

Convexity and Global Minimums: Any local minimum of a convex function will also be a global minimum. A strictly convex function will have at most a single minimum: the global minimum.

This means if we find a minimum, then we must have also found a global minimum (our goal).

6.2 Gradients

For a function $f(\alpha) = f(\alpha_1, \alpha_2, \ldots, \alpha_d)$, and a unit vector $u = (u_1, u_2, \ldots, u_d)$ which represents a direction, then the *directional derivative* is defined

$$\nabla_u f(\alpha) = \lim_{h \to 0} \frac{f(\alpha + hu) - f(\alpha)}{h}.$$

We are interested in functions f which are *differentiable*; this implies that $\nabla_u f(\alpha)$ is well-defined for all α and u. The converse is not necessarily true.

Let $e_1, e_2, \ldots, e_d \in \mathbb{R}^d$ be a specific set of unit vectors so that $e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ where for e_i the 1 is in the ith coordinate.

copyright: Jeff M. Phillips

Then define

$$\nabla_i f(\alpha) = \nabla_{e_i} f(\alpha) = \frac{\mathrm{d}}{\mathrm{d}\alpha_i} f(\alpha).$$

It is the derivative in the ith coordinate, treating all other coordinates as constants.

We can now, for a differentiable function f, define the *gradient of* f as

$$\nabla f = \frac{\mathrm{d}f}{\mathrm{d}\alpha_1} e_1 + \frac{\mathrm{d}f}{\mathrm{d}\alpha_2} e_2 + \ldots + \frac{\mathrm{d}f}{\mathrm{d}\alpha_d} e_d = \left(\frac{\mathrm{d}f}{\mathrm{d}\alpha_1}, \frac{\mathrm{d}f}{\mathrm{d}\alpha_2}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}\alpha_d} \right).$$

Note that ∇f is a function from $\mathbb{R}^d \to \mathbb{R}^d$, which we can evaluate at any point $\alpha \in \mathbb{R}^d$.

Example: Gradient

For $\alpha = (x, y, z) \in \mathbb{R}^3$, consider the function $f(x, y, z) = 3x^2 - 2y^3 - 2xe^z$. Then $\nabla f = (6x - 2e^z, -6y^2, -2xe^z)$ and $\nabla f(3, -2, 1) = (18 - 2e, 24, -6e)$.

Linear approximation. From the gradient we can easily recover the directional derivative of f at point α, for any direction (unit vector) u as

$$\nabla_u f(\alpha) = \langle \nabla f(\alpha), u \rangle.$$

This implies the gradient describes the linear approximation of f at a point α. The slope of the tangent plane of f at α in any direction u is provided by $\nabla_u f(\alpha)$.

Hence, the direction which f is increasing the most at a point α is the unit vector u where $\nabla_u f(\alpha) = \langle \nabla f(\alpha), u \rangle$ is the largest. This occurs at $\overline{\nabla f(\alpha)} = \nabla f(\alpha) / \|\nabla f(\alpha)\|$, the normalized gradient vector.

To find the minimum of a function f, we then typically want to move from any point α in the direction $-\overline{\nabla f(\alpha)}$; at regular points this is the direction of steepest descent.

6.3 Gradient Descent

Gradient descent is a family of techniques that, for a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$, try to identify either

$$\min_{\alpha \in \mathbb{R}^d} f(\alpha) \qquad \text{and/or} \qquad \alpha^* = \arg\min_{\alpha \in \mathbb{R}^d} f(\alpha).$$

This is effective when f is convex and we do not have a "closed form" solution α^*. The algorithm is iterative, in that it may never reach the completely optimal α^*, but it keeps getting closer and closer.

Algorithm 6.3.1 Gradient Descent$(f, \alpha_{\text{start}})$

initialize $\alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d$.
repeat
  $\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f(\alpha^{(k)})$
until $(\|\nabla f(\alpha^{(k)})\| \leq \tau)$
return $\alpha^{(k)}$

Basically, for any starting point $\alpha^{(0)}$ the algorithm moves to another point in the direction opposite to the gradient, in the direction that locally decreases f the fastest. How fast it moves depends on the scalar learning rate γ_k and the magnitude of the gradient vector $\nabla f(\alpha^{(k)})$.

Stopping condition. The parameter τ is the tolerance of the algorithm. If we assume the function is differentiable, then at the minimum α^*, we must have that $\nabla f(\alpha) = (0, 0, \ldots, 0)$. So for α close to the minimum, $\nabla f(\alpha)$ should also have a small norm. The algorithm may never reach the true minimum (and we do not know what it is, so we cannot directly compare against the function value). So we use $\|\nabla f\|$ as a proxy.

In other settings, we may run for a fixed number T steps. Although this does not automatically tune the algorithm to the input, as using a tolerance τ may, it is easier to describe and compare. Hence, most examples in this text will use this method.

6.3.1 Learning Rate

The most critical parameter of gradient descent is γ, the *learning rate*. In many cases the algorithm will keep $\gamma_k = \gamma$ fixed for all k. It controls how fast the algorithm works. But if it is too large, when we approach the minimum, then the algorithm may go too far, and overshoot it.

How should we choose γ? There is no consensus to this answer, and often in practice it is tuned in ad-hoc ways. In the following, we will describe some mathematically described scenarios where something formal can be said about how to choose γ. In some cases, these analysis show that if the function satisfies a mild property, then many fixed choices of γ will result in accuracy guarantees. We will also show methods where it helps to adjust the learning parameter γ_k adaptively.

Lipschitz bound. We say a function $g : \mathbb{R}^d \to \mathbb{R}^k$ is *L-Lipschitz* if for all $p, q \in \mathbb{R}^d$ that

$$\|g(p) - g(q)\| \leq L\|p - q\|.$$

This property is useful when $g = \nabla f$ is describing the gradient of a cost function f. If ∇f is L-Lipschitz, and we set $\gamma \leq \frac{1}{L}$, then gradient descent will converge to a stationary point. Moreover, if f is convex with global minimum α^*, then after $k = O(1/\varepsilon)$ steps we can guarantee that

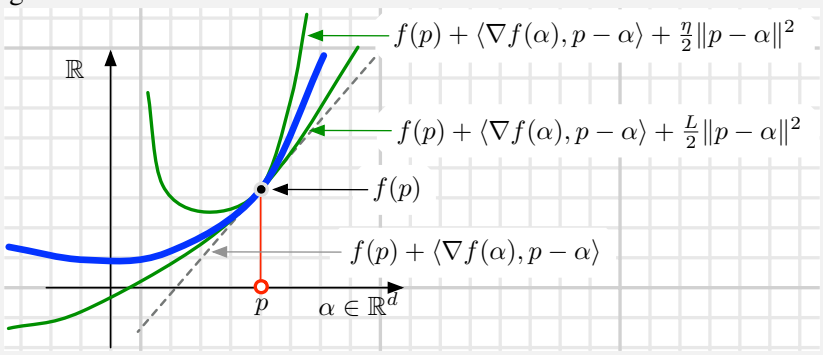
$$f(\alpha^{(k)}) - f(\alpha^*) \leq \varepsilon.$$

For the $k = O(1/\varepsilon)$ claim (and others stated below), we assume that $f(\alpha^{(0)}) - f(\alpha^*)$ is less than some absolute constant. Intuitively, the closer we start to the optimum, the fewer steps it will take.

For a convex quadratic function f (e.g., most cost functions derived by sum of squared errors), then the gradient ∇f is L-Lipschitz.

Example: Strongly Convex Function

We show an example η-strongly convex function f, in blue. At any point p, it is sandwiched between two convex quadratic functions in green. The convex quadratic function which lower bounds f has an L-Lipschitz gradient.



copyright: Jeff M. Phillips

Strongly Convex Functions. A function $f : \mathbb{R}^d \to \mathbb{R}$ is *η-strongly convex* with parameter $\eta > 0$ if for all $\alpha, p \in \mathbb{R}^d$ then

$$f(p) \leq f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{\eta}{2}\|p - \alpha\|^2.$$

Intuitively, this implies that f is at least quadratic. That is, along any direction $u = \frac{p-\alpha}{\|p-\alpha\|}$ the function $\langle f, u \rangle$ is 1-dimensional, and then its second derivative $\frac{\mathrm{d}^2}{\mathrm{d}u^2}\langle f, u \rangle$ is strictly positive; it is at least $\eta > 0$. Similarly, saying a function f has an L-Lipschitz gradient is equivalent to the condition that $\frac{\mathrm{d}^2}{\mathrm{d}u^2}\langle f(\alpha), u \rangle \leq L$ for all $\alpha, p \in \mathbb{R}^d$ where $u = \frac{p-\alpha}{\|p-\alpha\|}$.

For an η-strongly convex function f, that has an L-Lipschitz gradient, with global minimum α^*, then gradient descent with learning rate $\gamma \leq 2/(\eta + L)$ after only $k = O(\log(1/\varepsilon))$ steps will achieve

$$f(\alpha^{(k)}) - f(\alpha^*) \leq \varepsilon.$$

The constant in $k = O(\log(1/\varepsilon))$ depends on the condition number L/η. The conditions of this bound, imply that f is sandwiched between two convex quadratic functions; specifically for any $\alpha, p \in \mathbb{R}^d$ that we can bound

$$f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{L}{2}\|p - \alpha\|^2 \leq f(p) \leq f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{\eta}{2}\|p - \alpha\|^2.$$

When an algorithm converges at such a rate (takes $O(\log(1/\varepsilon))$ steps to obtain ε error), it is known as *linear convergence* since the log-error $\log(f(\alpha^{(k)}) - f(\alpha^*))$ looks like a linear function of k.

In practice, since many functions we consider will be convex quadratic functions (e.g., are derived from sum of squared error cost functions), then the error will decreases exponentially fast in terms of the number of steps of gradient descent, if the learning rate is set sufficiently small. That is, only a constant number of steps are required to resolve each bit of precision in the function value at the optimum! However, if the learning rate is set too small, then the constant (number of steps to resolve one bit) will increase.

So at this point, we have explained that for many situations there is a learning rate for which gradient descent will work extremely well. If we can analytically bound various properties of the second derivative of the function, then we can use these bounds to choose such a rate. However, we have not yet explained a formal way to find such a rate in general where we can only evaluate the gradient at any point $\alpha \in \mathbb{R}^d$.

Line search. An alternative, referred to as "line search" is to solve for the (approximately) optimal γ_k at each step. Once we have computed the gradient $\nabla f(\alpha^{(k)})$ then we have reduced the high-dimensional minimization problem to a one-dimensional problem. Note if f is convex, then f restricted to this one-dimensional search is also convex. We still need to find the minimum of an unknown function, but we can perform some procedure akin to binary search. We first find a value γ' such that

$$f\left(\alpha^{(k)} - \gamma' \nabla f(\alpha^{(k)})\right) > f(\alpha^{(k)})$$

then we keep subdividing the region $[0, \gamma']$ into pieces, and excluding ones which cannot contain the minimum.

For instance the *golden section search* divides a range $[b, t]$ containing the optimal γ_k into three subintervals (based on the golden ratio) so $[b, t] = [b, b') \cup [b', t'] \cup (t', t]$. And each step, we can determine that either $\gamma_k \notin [b, b')$ if $f(t') < f(b')$, or $\gamma_k \notin (t', t]$ if $f(b') < f(t')$. This reduces the range to $[b', t]$ or $[b, t']$, respectively, and we recurse.

In other situations, we can solve for the optimal γ_k exactly at each step. This is the case if we can again analytically take the derivative $\frac{\mathrm{d}}{\mathrm{d}\gamma}\left(f(\alpha^{(k)}) - \gamma \nabla f(\alpha^{(k)})\right)$ and solve for the γ where it is equal to 0.

 copyright: Jeff M. Phillips

Adjustable rate. In practice, line search is often slow. Also, we may not have a Lipschitz bound. It is often better to try a few fixed γ values, probably being a bit conservative. As long as $f(\alpha^{(k)})$ keep decreasing, it works well. This also may alert us if there is more than one local minimum if the algorithm converges to different locations.

An algorithm called "backtracking line search" automatically tunes the parameter γ. It uses a fixed parameter $\beta \in (0,1)$ (preferably in $(0.1, 0.8)$; for instance use $\beta = 3/4$). Start with a large step size γ (e.g., $\gamma = 1$). Then at each step of gradient descent at location α, if

$$f(\alpha - \gamma \nabla f(\alpha)) > f(\alpha) - \frac{\gamma}{2}\|\nabla f(\alpha)\|^2$$

then update $\gamma = \beta\gamma$. This shrinks γ over the course of the algorithm, and if f is strongly convex, it will eventually decrease γ until it satisfies the condition for linear convergence.

Example: Gradient Descent with Fixed Learning Rate

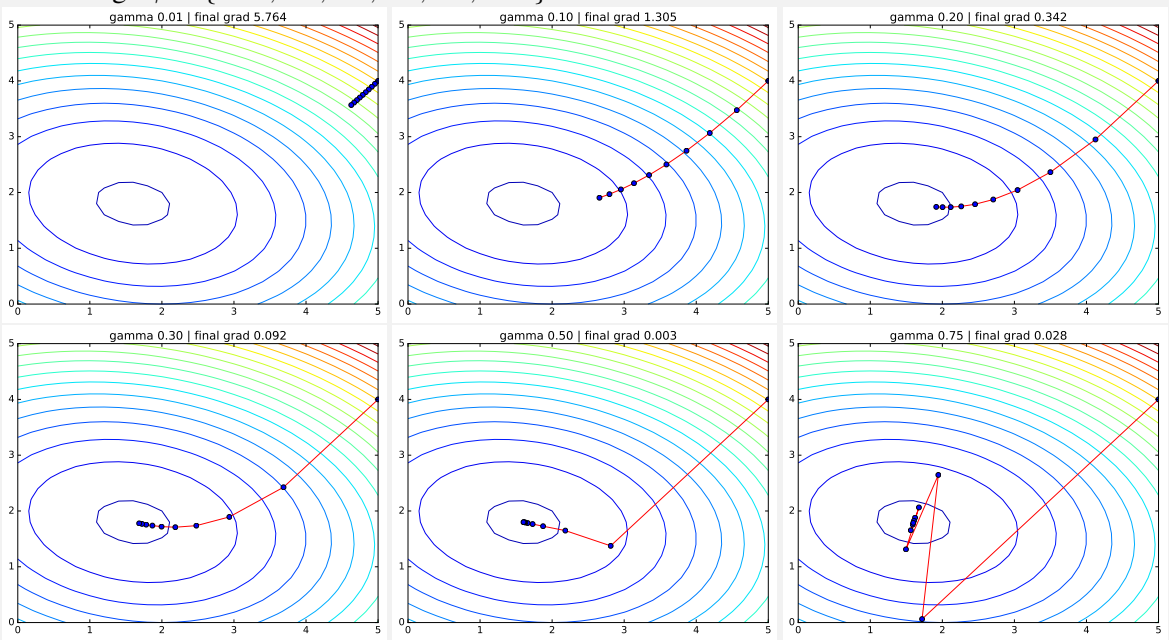
Consider the function f where $\alpha = (x, y) \in \mathbb{R}^2$ is defined

$$f(x,y) = (\frac{3}{4}x - \frac{3}{2})^2 + (y-2)^2 + \frac{1}{4}xy$$

and has gradient

$$\nabla f(x,y) = \left(\frac{9}{8}x - \frac{9}{4} + \frac{1}{4}y \;\;,\;\; 2y - 4 + \frac{1}{4}x\right).$$

We run gradient descent for 10 iterations within initial position $(5, 4)$, while varying the learning rate in the range $\gamma = \{0.01, 0.1, 0.2, 0.3, 0.5, 0.75\}$.



We see that with γ very small, the algorithm does not get close to the minimum. When γ is too large, then the algorithm jumps around a lot, and is in danger of not converging. But at a learning rate of $\gamma = 0.3$ it converges fairly smoothly and reaches a point where $\|\nabla f(x,y)\|$ is very small. Using $\gamma = 0.5$ almost overshoots in the first step; $\gamma = 0.3$ is smoother, and it is probably best to use a curve that looks smooth like that one, but with a few more iterations.

 copyright: Jeff M. Phillips

```
import matplotlib as mpl
mpl.use('PDF')
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA

def func(x,y):
  return (0.75*x-1.5)**2 + (y-2.0)**2 + 0.25*x*y

def func_grad(vx,vy):
  dfdx = 1.125*vx - 2.25 + 0.25*vy
  dfdy = 2.0*vy - 4.0 + 0.25*vx
  return np.array([dfdx,dfdy])

#prepare for contour plot
xlist = np.linspace(0, 5, 26)
ylist = np.linspace(0, 5, 26)
x, y = np.meshgrid(xlist, ylist)
z = func(x,y)
lev = np.linspace(0,20,21)

#iterate location
v_init = np.array([5,4])
num_iter = 10
values = np.zeros([num_iter,2])

for gamma in [0.01, 0.1, 0.2, 0.3, 0.5, 0.75]:
  values[0,:] = v_init
  v = v_init

  # actual gradient descent algorithm
  for i in range(1,num_iter):
    v = v - gamma * func_grad(v[0],v[1])
    values[i,:] = v

  #plotting
  plt.contour(x,y,z,levels=lev)
  plt.plot(values[:,0],values[:,1],'r-')
  plt.plot(values[:,0],values[:,1],'bo')
  grad_norm = LA.norm(func_grad(v[0],v[1]))
  title = "gamma %0.2f | final grad %0.3f" % (gamma,grad_norm)
  plt.title(title)
  file = "gd-%2.0f.pdf" % (gamma*100)
  plt.savefig(file, bbox_inches='tight')
  plt.clf()
  plt.cla()
```

6.4 Fitting a Model to Data

For data analysis, the most common use of gradient descent is to fit a model to data. In this setting we have a data set $(X, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \in \mathbb{R}^d \times \mathbb{R}$ and a family of models $\mathcal{M}$ so each possible model M_α is defined by a d-dimensional vector $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ for k parameters.

Next we define a *loss function* $L((X, y), M_\alpha)$ which measures the difference between what the model predicts and what the data values are. To choose which parameters generate the best model, we let $f(\alpha) : \mathbb{R}^d \to \mathbb{R}$ be our function of interest, defined $f(\alpha) = L((X, y), M_\alpha)$. Then we can run gradient descent to

find our model M_{α^*}. For instance we can set

$$f(\alpha) = L((X,y), M_\alpha) = \mathsf{SSE}(P, M_\alpha) = \sum_{(x_i,y_i)\in(X,y)} (y_i - M_\alpha(x_i))^2. \tag{6.1}$$

This is used for examples including maximum likelihood (or maximum log-likelihood) estimators from Bayesian inference. This includes finding a single point estimator with Gaussian (where we had a closed-form solution), but also many other variants (often where there is no known closed-form solution). It also includes least squares regression and its many variants; we will see this in much more detail next. And will include other topics (including clustering, PCA, classification) we will see later in class.

6.4.1 Least Mean Squares Updates for Regression

Now we will work through how to use gradient descent for simple quadratic regression on 1-dimensional explanatory variables. That is, this will specify the function $f(\alpha)$ in equation (6.1) to have $k = 3$ parameters as $\alpha = (\alpha_0, \alpha_1, \alpha_2)$, and so for each $(x_i, y_i) \in (X, y)$ we have $x_i \in \mathbb{R}$. Then we can write the model again as a dot product

$$M_\alpha(x_i) = \langle \alpha, (1, x_i, x_i^2)\rangle = \alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2$$

It is straightforward to generalize to linear regression, multiple-explanatory variable linear regression, or general polynomial regression from here. For instance, in this case, we can represent x_i as the 3-dimensional feature vector $q = (q_0 = x_i^0 = 1, q_1 = x_i^1, q_2 = x_i^2)$ of explanatory variables. Then the model is $M_\alpha(x_i) = \langle \alpha, q\rangle$. For simple linear regression, we simply omit the quadratic term $\alpha_2 x_i^2$ from the above model. For other polynomial forms of regression, the expansion simply includes more terms and powers. And for multiple explanatory variables they are each given a corresponding feature coordinate, possible to some power, or multiplied with other explanatory variables in the polynomial models. We will continue using 1-dimensional quadratic regression as an example.

Now to specify the gradient descent step

$$\alpha := \alpha - \gamma \nabla f(\alpha)$$

we only need to define $\nabla f(\alpha)$. We will first show this for the case where $n = 1$, that is when there is a single data point (x_1, y_1). For quadratic regression, the cost function $f_1(\alpha) = (\alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 - y_1)^2$ is convex. Next derive

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\alpha_j} f(\alpha) &= \frac{\mathrm{d}}{\mathrm{d}\alpha_j}(M_\alpha(x_1) - y_1)^2 \\
&= 2(M_\alpha(x_1) - y_1)\frac{\mathrm{d}}{\mathrm{d}\alpha_j}(M_\alpha(x_1) - y_1) \\
&= 2(M_\alpha(x_1) - y_1)\frac{\mathrm{d}}{\mathrm{d}\alpha_j}(\sum_{j=0}^{2} \alpha_j x_1^j - y_1) \\
&= 2(M_\alpha(x_1) - y_1)x_1^j
\end{aligned}$$

Using this convenient form (which generalizes to any polynomial model), we define

$$\begin{aligned}
\nabla f(\alpha) &= \left(\frac{\mathrm{d}}{\mathrm{d}\alpha_0} f(\alpha), \frac{\mathrm{d}}{\mathrm{d}\alpha_1} f(\alpha), \frac{\mathrm{d}}{\mathrm{d}\alpha_2} f(\alpha)\right) \\
&= 2\left((M_\alpha(x_1) - y_1),\ \ (M_\alpha(x_1) - y_1)x_1,\ \ (M_\alpha(x_1) - y_1)x_1^2\right).
\end{aligned}$$

Applying $\alpha := \alpha - \gamma \nabla f(\alpha)$ according to this specification is known as the *LMS (least mean squares)* update rule or the *Widrow-Huff learning rule*. Quite intuitively, the magnitude the update is proportional to the residual norm $(M_\alpha(x_1) - y_1)$. So if we have a lot of error in our guess of α, then we take a large step; if we do not have a lot of error, we take a small step.

6.4.2 Decomposable Functions

To generalize this to multiple data points ($n > 1$), there are two standard ways. Both of these take strong advantage of the cost function $f(\alpha)$ being *decomposable*. That is, we can write

$$f(\alpha) = \sum_{i=1}^{n} f_i(\alpha),$$

where each f_i depends only on the ith data point $p_i \in P$. In particular, where $p_i = (x_i, y_i)$, then for quadratic regression

$$f_i(\alpha) = (M_\alpha(x_i) - y_i)^2 = (\alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2 - y_i)^2.$$

First notice that since f is the sum of f_is, where each is convex, then f must also be convex; in fact the sum of these usually becomes strongly convex (as long as the corresponding feature vectors are full rank). Also two approaches towards gradient descent will take advantage of this decomposition in slightly different ways. This decomposable property holds for most loss functions for fitting a model to data.

Batch gradient descent. The first technique, called *batch gradient descent*, simply extends the definition of $\nabla f(\alpha)$ to the case with multiple data points. Since f is decomposable, then we use the linearity of the derivative to define

$$\frac{\mathrm{d}}{\mathrm{d}\alpha_j} f(\alpha) = \sum_{i=1}^{n} \frac{\mathrm{d}}{\mathrm{d}\alpha_j} f_i(\alpha) = \sum_{i=1}^{n} 2(M_\alpha(x_i) - y_i)x_i^j$$

and thus

$$\begin{aligned}
\nabla f(\alpha) &= \left(2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i),\ \ 2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i)x_i,\ \ 2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i)x_i^2 \right) \\
&= \sum_{i=1}^{n} \left(2(M_\alpha(x_i) - y_i),\ \ 2(M_\alpha(x_i) - y_i)x_i,\ \ 2(M_\alpha(x_i) - y_i)x_i^2 \right) \\
&= 2\sum_{i=1}^{n}(M_\alpha(x_i) - y_i)\left(1, x_i, x_i^2\right).
\end{aligned}$$

That is, the step is now just the sum of the terms from each data point. Since f is (strongly) convex, then we can apply all of the nice convergence results discussed about (strongly) convex f before. However, computing $\nabla f(\alpha)$ each step takes $O(n)$ time, which can be slow for large n (i.e., for large data sets).

Algorithm 6.4.1 Incremental Gradient Descent$(f, \alpha_{\text{start}})$

initialize $\alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d$; $i = 1$.
repeat
  $\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f_i(\alpha^{(k)})$
  $i := (i + 1) \mod n$
until $(\|\nabla f(\alpha^{(k)})\| \leq \tau)$ $\quad (\star)$
return $\alpha^{(k)}$

Stochastic gradient descent. The second technique is called *incremental gradient descent*; see Algorithm 6.4.1. It avoids computing the full gradient each step, and only computes $\nabla f_i(\alpha)$ for a single data point $p_i \in P$. For quadratic regression with $M_\alpha(x) = \langle \alpha, (1, x, x^2) \rangle$ it is

$$\nabla f_i(\alpha) = 2(M_\alpha(x_i) - y_i)(1, x_i, x_i^2).$$

Implementation Hints:
($\star$) The norm of a single gradient is not stable. Instead a better stopping condition averages the gradient norm over several (lets say B steps). The condition may then be $\frac{1}{B}\sum_{b=0}^{B-1} \|\nabla f(\alpha^{(k-b)})\| \leq \tau$, and is only checked after B steps are taken.

A more common variant of this is called *stochastic gradient descent*; see Algorithm 6.4.2. Instead of choosing the data points in order, it selects a data point p_i at random each iteration (the term "stochastic" refers to this randomness).

Algorithm 6.4.2 Stochastic Gradient Descent$(f, \alpha_{\text{start}})$

initialize $\alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d$
repeat
  Randomly choose $i \in \{1, 2, \ldots, n\}$
  $\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f_i(\alpha^{(k)})$
until $(\|\nabla f(\alpha^{(k)})\| \leq \tau)$ ($\star$)
return $\alpha^{(k)}$

On very large data sets (i.e., big data!), these algorithms are often much faster than the batch version since each iteration now takes $O(1)$ time. However, it does not automatically inherit all of the nice convergence results from what is known about (strongly) convex functions. Yet in many settings, there is an abundance of data points described by the same model. They should have a roughly similar effect. In practice when one is far from the optimal model, these steps converge about as well as the batch version (but much much faster in runtime). When one is close to the optimal model, then the incremental / stochastic variants may not exactly converge. However, if one is satisfied to reach a point that is close enough to optimal, there are some randomized (PAC-style) guarantees possible for the stochastic variant. And in fact, for very large data sets (i.e., n is very big) they typically converge before the algorithm even uses all (or even most) of the data points.

 copyright: Jeff M. Phillips

Exercises

We will use a dataset `http://www.cs.utah.edu/~jeffp/teaching/FoDA/D4.csv`

Q6.1: Consider a function $f(x, y)$ with gradient $\nabla f(x, y) = (x - 1, 2y + x)$. Starting at a value $(x = 1, y = 2)$, and a learning rate of $\gamma = 1$, execute one step of gradient descent.

Q6.2: Consider running gradient descent with a fixed learning rate γ. For each of the following, we plot the function value over 10 steps (the function is different each time). Decide whether the learning rate is probably **too high**, **too low**, or **about right**.

1. f_1: 100, 99, 98, 97, 96, 95, 94, 93, 92, 91
2. f_2: 100, 50, 75, 60, 65, 45, 75, 110, 90, 85
3. f_3: 100, 80, 65, 50, 40, 35, 31, 29, 28, 27.5, 27.3
4. f_4: 100, 80, 60, 40, 20, 0, -20, -40, -60, -80, -100

Q6.3: Consider two functions

$$f_1(x, y) = (x - 5)^2 + (y + 2)^2 - 2xy \qquad f_2(x, y) = (1 - (y - 4))^2 + 20((x + 6) - (y - 4)^2)^2$$

Starting with $(x, y) = (0, 2)$ run the gradient descent algorithm for each function. Run for T iterations, and report the function value at the end of each step.

1. First, run with a fixed learning rate of $\gamma = 0.05$ for f_1 and $\gamma = 0.0015$ for f_2.
2. Second, run with any variant of gradient descent you want. Try to get the smallest function value after T steps.

For f_1 you are allowed only $T = 10$ steps. For f_2 you are allowed $T = 100$ steps.

Q6.4: In the first `D4.csv` dataset provided, use the first three columns as explanatory variables x_1, x_2, x_3, and the fourth as the dependent variable y. Run gradient descent on $\alpha \in \mathbb{R}^4$, using the dataset provided to find a linear model

$$\hat{y} = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3$$

minimizing the sum of squared errors. Run for as many steps as you feel necessary. On each step of your run, print on a single line: (1) the model parameters $\alpha^{(i)} = [\alpha_0^{(i)}, \alpha_1^{(i)}, \alpha_2^{(i)}, \alpha_3^{(i)}]$, (2) the value of a function $f(\alpha^{(i)})$, estimating the sum of squared errors, and (3) the gradient $\nabla f(\alpha^{(i)})$. *(These are the sort of things you would do to check/debug a gradient descent algorithm; you may also want to plot some of these.)*

1. First run batch gradient descent.
2. Second run incremental gradient descent.

Choose one method which you preferred (either is ok to choose), and explain why you preferred it to the other method.

Q6.5: Explain what parts of the above procedures would change if you instead are minimizing the sum of residuals, not the sum of squared residuals?

- Is the function still convex?
- Does the gradient always exist?

 copyright: Jeff M. Phillips

7 Principal Component Analysis

This topic will build a series of techniques to deal with high-dimensional data. Unlike regression problems, our goal is not to predict a value (the y-coordinate), it is to understand the "shape" of the data, for instance a low-dimensional representation that captures most of meaning of the high-dimensional data. This is sometimes referred to as *unsupervised learning* (as opposed to regression and classification, where the data has labels, known as supervised learning). Like most unsupervised settings, it can be a lot of fun, but its easy to get yourself into trouble if you are not careful.

We will cover many interconnected tools, including the singular value decomposition (SVD), eigenvectors and eigenvalues, the power method, principal component analysis, and multidimensional scaling.

7.1 Data Matrices

We will start with data in a matrix $A \in \mathbb{R}^{n \times d}$, and will call upon linear algebra to rescue us. It is useful to think of each row a_i of A as a data point in $\mathbb{R}^d$, so there are n data points. Each dimension $j \in 1, 2, \ldots, d$ corresponds with an attribute of the data points.

Example: Data Matrices

There are many situations where data matrices arise.

- Consider a set of n weather stations reporting temperature over d points in time. Then each row a_i corresponds to a single weather station, and each coordinate $A_{i,j}$ of that row is the temperature at station i at time j.
- In movie ratings, we may consider n users who have rated each of d movies on a score of $1-5$. Then each row a_i represents a user, and the jth entry of that user is the score given to the j movie.
- Consider the price of a stock measured over time (say the closing price each day). Many time-series models consider some number of days (d days, for instance 25 days) to capture the pattern of the stock at any given time. So for a given closing day, we consider the d previous days. If we have data on the stock for 4 years (about 1000 days the stock market is open), then we can create a d-dimensional data points (the previous $d = 25$ days) for each day (except the first 25 or so). The data matrix is then comprised of n data points a_i, where each corresponds to the closing day, and the previous d days. The jth entry is the value on $(j-1)$ days before the closing day i.
- Finally consider a series of pictures of a shape (say the Utah teapot). The camera position is fixed as is the background, but we vary two things: the rotation of the teapot, and the amount of light. Here each pictures is a set of say d pixels (say 10,000 if it is 100×100), and there are n pictures. Each picture is a row of length d, and each pixel corresponds to a column of the matrix. Similar, but more complicated scenarios frequently occur with pictures of a persons face, or $3d$-imaging of an organ.

In each of these scenarios, there are many (n) data points, each with d attributes. The following will be very important:

- **all coordinates have the same units!**

If this "same units" property does not hold, then when we measure a distance between data points in $\mathbb{R}^d$, usually using the L_2-norm, then the distance is nonsensical.

The next goal is to uncover a pattern, or a model M. In this case, the model will be a low-dimensional subspace F. It will represent a k-dimensional space, where $k << d$. For instance in the example with images, there are only two parameters which are changing (rotation, and lighting), so despite having $d = 10{,}000$ dimensions of data, 2 should be enough to represent everything.

7.1.1 Projections

Different than in linear regression this family of techniques will measure error as a projection from $a_i \in \mathbb{R}^d$ to the closest point $\pi_F(a_i)$ on F. To define this we will use linear algebra.

First recall, that given a unit vector $u \in \mathbb{R}^d$ and any data point $p \in \mathbb{R}^d$, then the dot product

$$\langle u, p \rangle$$

is the norm of p projected onto the line through u. If we multiply this scalar by u then

$$\pi_u(p) = \langle u, p \rangle u,$$

and it results in the point on the line through u that is closest to data point p. This is a *projection of p onto u*.

To understand this for a subspace F, we will need to define a basis. *For now we will assume that F contains the origin $(0, 0, 0, \ldots, 0)$ (as did the line through u).* Then if F is k-dimensional, then this means there is a k-dimensional basis $U_F = \{u_1, u_2, \ldots, u_k\}$ so that

- For each $u_i \in U_F$ we have $\|u_i\| = 1$, that is u_i is a unit vector.
- For each pair $u_i, u_j \in U_F$ we have $\langle u_i, u_j \rangle = 0$; the pairs are orthogonal.
- For any point $x \in F$ we can write $x = \sum_{i=1}^{k} \alpha_i u_i$; in particular $\alpha_i = \langle x, u_i \rangle$.

Given such a basis, then the projection on to F of a point $p \in \mathbb{R}^d$ is simply

$$\pi_F(p) = \sum_{i=1}^{k} \langle u_i, p \rangle u_i.$$

Thus if p happens to be exactly in F, then this recovers p exactly.

The other powerful part of the basis U_F is the it defines a *new coordinate system*. Instead of using the d original coordinates, we can use new coordinates $(\alpha_1(p), \alpha_2(p), \ldots, \alpha_k(p))$ where $\alpha_i(p) = \langle u_i, p \rangle$. To be clear $\pi_F(p)$ is still in $\mathbb{R}^d$, but there is a k-dimensional representation if we restrict to F.

When F is d-dimensional, this operation can still be interesting. The basis we choose $U_F = \{u_1, u_2, \ldots, u_d\}$ could be the same as the original coordinate axis, that is we could have $u_i = e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ where only the ith coordinate is 1. But if it is another basis, then this acts as a rotation (with possibility of also a mirror flip). The first coordinate is rotated to be along u_1; the second along u_2; and so on. In $\pi_F(p)$, the point p does not change, just its representation.

7.1.2 SSE Goal

As usual our goal will be to minimize the sum of squared errors. In this case we define this as

$$\mathsf{SSE}(A, F) = \sum_{a_i \in A} \|a_i - \pi_F(a_i)\|^2,$$

and our desired k-dimensional subspace F is

$$F^* = \arg\min_F \mathsf{SSE}(A, F)$$

As compared to linear regression, this is much less a "proxy goal" where the true goal was prediction. Now we have no labels (the y_i values), so we simply try to fit a model through all of the data.

How do we solve for this?

- Linear regression does not work, its cost function is different.
- It is not obvious how to use gradient descent. The restriction that each $u_i \in U_F$ is a unit vector puts in a constraint, in fact a non-convex one. There are ways to deal with this, but we have not discussed these yet.
- ... linear algebra will come back to the rescue, now in the form of the SVD.

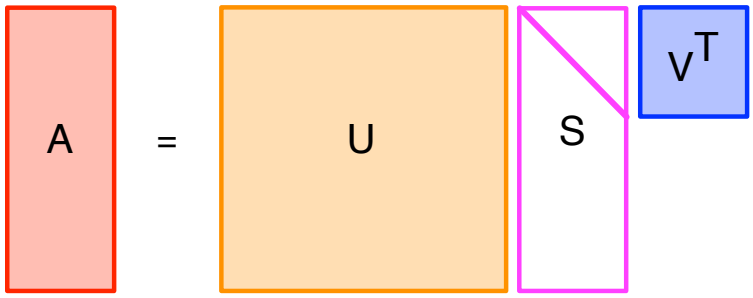
7.2 Singular Value Decomposition

A really powerful and useful linear algebra operation is called the *singular value decomposition*. It extracts an enormous amount of information about a matrix A. This section will define it and discuss many of its uses. Then we will describe one algorithm how to construct it. But in general, one simply calls the procedure in your favorite programming language and it calls the same highly-optimized back-end from the Fortran LAPACK library.

```
from scipy import linalg as LA
U, s, Vt = LA.svd(A)
```

The SVD takes in a matrix $A \in \mathbb{R}^{n \times d}$ and outputs three matrices $U \in \mathbb{R}^{n \times n}$, $S \in \mathbb{R}^{n \times d}$ and $V \in \mathbb{R}^{d \times d}$, so that $A = USV^T$.

$$[U, S, V] = \mathsf{svd}(A)$$



The structure that lurks beneath. The matrix S only has non-zero elements along its diagonal. So $S_{i,j} = 0$ if $i \neq j$. The remaining values $\sigma_1 = S_{1,1}$, $\sigma_2 = S_{2,2}$, ..., $\sigma_r = S_{r,r}$ are known as the *singular values*. They have the property that

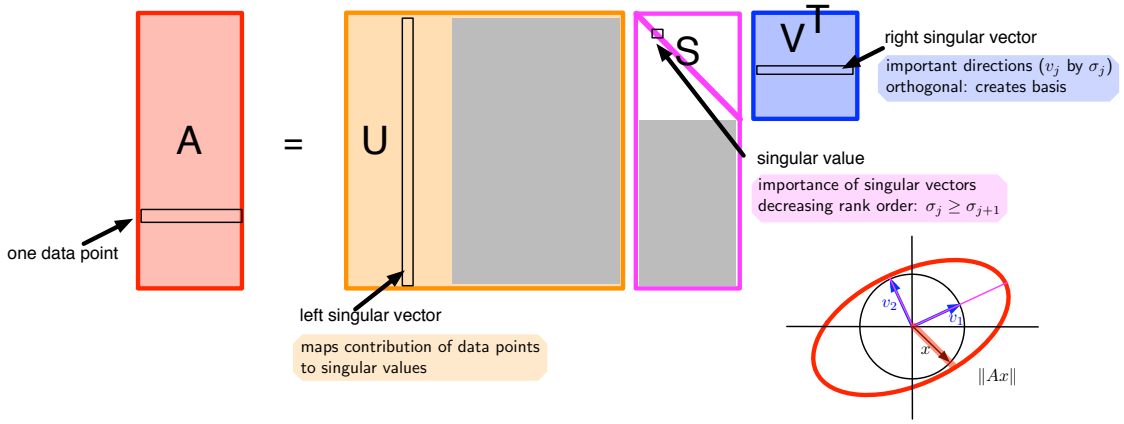
$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq 0$$

 copyright: Jeff M. Phillips

where $r \leq \min\{n, d\}$ is the rank of the matrix A. So the number of non-zero singular values reports the rank (this is a numerical way of computing the rank or a matrix).

The matrices U and V are orthogonal. Thus, their columns are all unit vectors and orthogonal to each other (within each matrix). The columns of U, written $u_1, u_2, \ldots, u_n$, are called the *left singular vectors*; and the columns of V (i.e., rows of V^T), written $v_1, v_2, \ldots, v_d$, are called the *right singular vectors*.

This means for any vector $x \in \mathbb{R}^d$, the columns of V (the right singular vectors) provide a basis. That is, we can write $x = \sum_{i=1}^{d} \alpha_i v_i$ for $\alpha_i = \langle x, v_i \rangle$. Similarly for any vector $y \in \mathbb{R}^n$, the columns of U (the left singular vectors) provide a basis. This also implies that $\|x\| = \|V^T x\|$ and $\|y\| = \|yU\|$.



Tracing the path of a vector. To illustrate what this decomposition demonstrates, a useful exercise is to trace what happens to a vector $x \in \mathbb{R}^d$ as it is left-multiplied by A, that is $Ax = USV^T x$.

First $V^T x$ produces a new vector $\xi \in \mathbb{R}^d$. It essentially changes no information, just changes the basis to that described by the right singular values. For instance the new i coordinate $\xi_i = \langle v_i, x \rangle$.

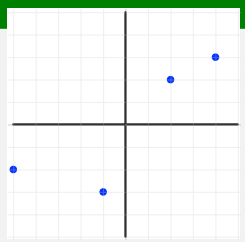
Next $\eta \in \mathbb{R}^n$ is the result of $SV^T x = S\xi$. It scales ξ by the singular values of S. Note that if $d < n$ (the case we will focus on), then the last $n - d$ coordinates of η are 0. In fact, for $j > r$ (where $r = \mathsf{rank}(A)$) then $\eta_j = 0$. For $j \leq r$, then the vector η is stretched longer in the first coordinates since these have larger values.

The final result is a vector $y \in \mathbb{R}^n$, the result of $Ax = USV^T x = U\eta$. This again just changes the basis of η so that it aligns with the left singular vectors. In the setting $n > d$, the last $n - d$ left singular vectors are meaningless since the corresponding entries in η are 0.

Working backwards ... this final U matrix can be thought of mapping the effect of η onto each of the data points of A. The η vector, in turn, can be thought of as scaling by the content of the data matrix A (the U and V^T matrices contain no scaling information). And the ξ vector arises via the special rotation matrix V^T that puts the starting point x into the right basis to do the scaling (from the original d-dimensional coordinates to one that suits the data better).

copyright: Jeff M. Phillips

Example: Tracing through the SVD



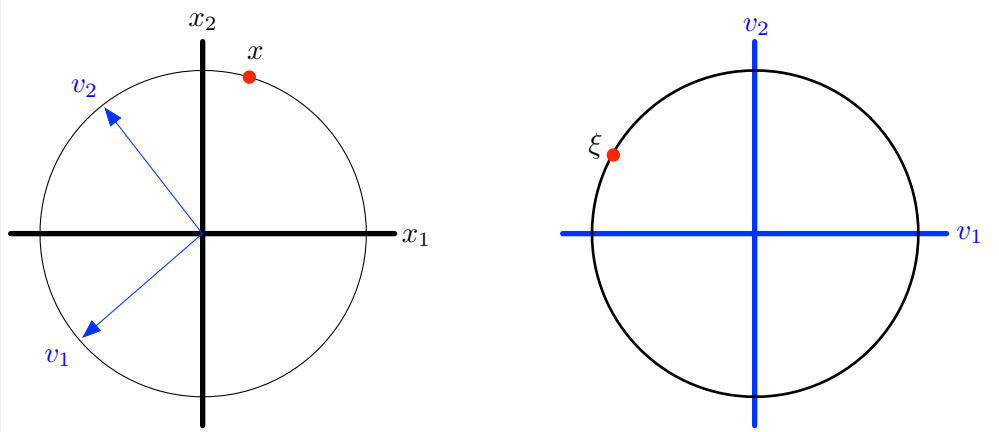
Consider a matrix

$$A = \begin{pmatrix} 4 & 3 \\ 2 & 2 \\ -1 & -3 \\ -5 & -2 \end{pmatrix},$$

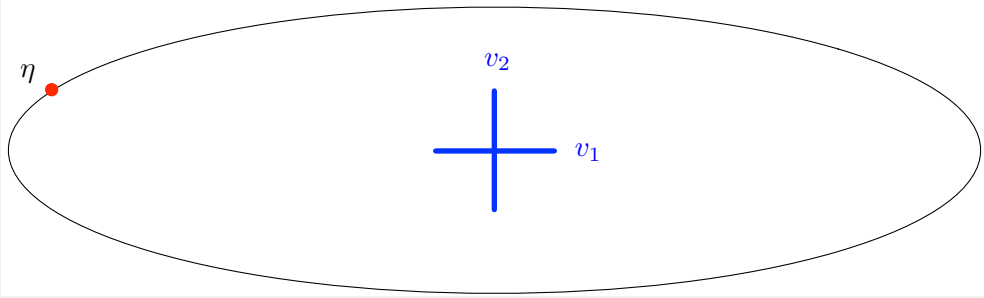
and its SVD $[U, S, V] = \mathsf{svd}(A)$:

$$U = \begin{pmatrix} -0.6122 & 0.0523 & 0.0642 & 0.7864 \\ -0.3415 & 0.2026 & 0.8489 & -0.3487 \\ 0.3130 & -0.8070 & 0.4264 & 0.2625 \\ 0.6408 & 0.5522 & 0.3057 & 0.4371 \end{pmatrix}, \quad S = \begin{pmatrix} 8.1655 & 0 \\ 0 & 2.3074 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} -0.8142 & -.5805 \\ -0.5805 & 0.8142 \end{pmatrix}.$$

Now consider a vector $x = (0.243, 0.97)$ (scaled very slightly so it is a unit vector, $\|x\| = 1$). Multiplying by V^T rotates (and flips) x to $\xi = V^T x$; still $\|\xi\| = 1$



Next multiplying by S scales ξ to $\eta = S\xi$. Notice there are an imaginary third and fourth coordinates now; they are both coming out of the page! Don't worry, they won't poke you since their magnitude is 0.



Finally, $y = U\eta = Ax$ is again another rotation of η in this four dimensional space.

copyright: Jeff M. Phillips

```
import scipy as sp
import numpy as np
from scipy import linalg as LA


A = np.array([[4.0,3.0], [2.0,2.0], [-1.0,-3.0], [-5.0,-2.0]])

U, s, Vt = LA.svd(A, full_matrices=False)

print U
#[[-0.61215255 -0.05228813]
# [-0.34162337 -0.2025832 ]
# [ 0.31300005  0.80704816]
# [ 0.64077586 -0.55217683]]
print s
#[ 8.16552039  2.30743942]
print Vt
#[[-0.81424526 -0.58052102]
# [ 0.58052102 -0.81424526]]

x = np.array([0.243,0.97])
x = x/LA.norm(x)

xi = Vt.dot(x)
print xi
#[-0.7609864  -0.64876784]

S = LA.diagsvd(s,2,2)
eta = S.dot(xi)
print eta
#[-6.21384993 -1.49699248]

y = U.dot(eta)
print y
#[ 3.88209899  2.42606187 -3.1530804  -3.15508046]

print A.dot(x)
#[ 3.88209899  2.42606187 -3.1530804  -3.15508046]
```

7.2.1 Best Rank-k Approximation

So how does this help solve the initial problem of finding F^*, which minimized the SSE? The singular values hold the key.

It turns out that there is a *unique* singular value decomposition, up to ties in the singular values. This means, there is exactly one (up to singular value ties) set of right singular values which rotate into a basis so that $\|Ax\| = \|SV^T x\|$ for all $x \in \mathbb{R}^d$ (recall that U is orthogonal, so it does not change the norm, $\|U\eta\| = \|\eta\|$).

Next we realize that the singular values come in sorted order $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$. In fact, they are defined so that we choose v_1 so it maximizes $\|Av_1\|$, then we find the next singular vector v_2 which is orthogonal to v_1 and maximizes $\|Av_2\|$, and so on. Then $\sigma_i = \|Av_i\|$.

If we define F with the basis $U_F = \{v_1, v_2, \ldots, v_k\}$, then

$$\|x - \pi_F(x)\|^2 = \left\| \sum_{i=1}^{d} v_i \langle x, v_i \rangle - \sum_{i=1}^{k} v_i \langle x, v_i \rangle \right\|^2 = \sum_{i=k+1}^{d} \langle x, v_i \rangle^2.$$

so the projection error is that part of x in the last $(d-k)$ right singular vectors.

But we are not trying to directly predict new data here (like in regression). Rather, we are trying to approximate the data we have. We want to minimize $\sum_i \|a_i - \pi_F(a_i)\|^2$. But for any unit vector u, we recall now that

$$\|Au\|^2 = \sum_{i=1}^{n} \langle a_i, u \rangle.$$

Thus the projection error can be measured with a set of orthonormal vectors $w_1, w_2, \ldots, w_{d-k}$ which are each orthogonal to F, as $\sum_{j=1}^{n-k} \|Aw_j\|^2$. When defining F as the first k right singular values, then these orthogonal vectors are the remaining $(n-k)$ right singular vectors, so the projection error is

$$\sum_{i=1}^{n} \|a_i - \pi_F(a_i)\|^2 = \sum_{j=k+1}^{d} \|Av_j\|^2 = \sum_{j=k+1}^{d} \sigma_j^2.$$

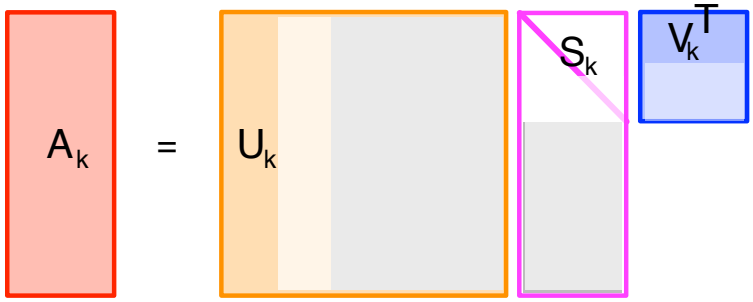
And thus by how the right singular vectors are defined, this expression is minimized when F is defined as the span of the first k singular values.

Best rank-k approximation. A similar goal is to find the *best rank-k approximation of A*. That is a matrix $A_k \in \mathbb{R}^{n \times d}$ so that $\mathsf{rank}(A_k) = k$ and it minimizes both

$$\|A - A_k\|_2 \quad \text{and} \quad \|A - A_k\|_F.$$

Note that $\|A - A_k\|_2 = \sigma_{k+1}$ and $\|A - A_k\|_F^2 = \sum_{j=k+1}^{d} \sigma_j^2$.

Remarkably, this A_k matrix also comes from the SVD. If we set S_k as the matrix S in the decomposition so that all but the first k singular values are 0, then it has rank k. Hence $A_k = US_kV^T$ also has rank k and is our solution. But we can notice that when we set most of S_k to 0, then the last $(d-k)$ columns of V are meaningless since they are only multiplied by 0s in US_kV^T, so we can also set those to all 0s, or remove them entirely (along with the last $(d-k)$ columns of S_k). Similar we can make 0 or remove the last $(n-k)$ columns of U. These matrices are referred to as V_k and U_k respectively, and also $A_k = U_kS_kV_k^T$.



7.3 Eigenvalues and Eigenvectors

A related matrix decomposition to SVD is the eigendecomposition. This is only defined for a square matrix $B \in \mathbb{R}^{n \times n}$.

An *eigenvector* of B is a vector v such that there is some scalar λ that

$$Bv = \lambda v.$$

That is, multiplying B by v results in a scaled version of v. The associated value λ is called the *eigenvalue*. As a convention, we typically normalize v so $\|v\| = 1$.

In general, a square matrix $B \in \mathbb{R}^{n \times n}$ may have up to n eigenvectors (a matrix $\mathtt{V} \in \mathbb{R}^{n \times n}$) and values (a vector $\mathtt{l} \in \mathbb{R}^n$). Some of the eigenvalues may be complex numbers (even when all of its entries are real!).

 copyright: Jeff M. Phillips

```
from scipy import linalg as LA
l, V = LA.eig(B)
```

For this reason, we will focus on positive semidefinite matrices. A *positive definite matrix* $B \in \mathbb{R}^{n \times n}$ is a symmetric matrix with all positive eigenvalues. Another characterization is for every vector $x \in \mathbb{R}^n$ then $x^T B x$ is positive. A *positive semidefinite matrix* $B \in \mathbb{R}^{n \times n}$ may have some eigenvalues at 0 and are otherwise positive; equivalently for any vector $x \in \mathbb{R}^n$, then $x^T B x$ may be zero or positive.

How do we get positive semi-definite matrices? Lets start with a data matrix $A \in \mathbb{R}^{n \times d}$. Then we can construct two positive semidefinite matrices

$$B_R = A^T A \qquad \text{and} \qquad B_L = A A^T.$$

Matrix B_R is $d \times d$ and B_L is $n \times n$. If the rank of A is d, then B_R is positive definite. If the rank of A is n, then B_L is positive definite.

Eigenvectors and eigenvalues relation to SVD. Next consider the SVD of A so that $[U, S, V] = \mathsf{svd}(A)$. Then we can write

$$B_R V = A^T A V = (V S U^T)(U S V^T) V = V S^2.$$

Note that the last step follows because for orthogonal matrices U and V, then $U^T U = I$ and $V^T V = I$, where I is the identity matrix, which has no effect. The matrix S is a diagonal square[1] matrix $S = \mathsf{diag}(\sigma_1, \sigma_2, \ldots, \sigma_d)$. Then $S^2 = SS$ (the product of S with S) is again diagonal with entries $S^2 = \mathsf{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2)$.

Now consider a single column v_i of V (which is the ith right singular vector of A). Then extracting this column's role in the linear system $B_R V = V S^2$ we obtain

$$B_R v_i = v_i \sigma_i^2.$$

This means that ith right singular vector of A is an eigenvector (in fact the ith eigenvector) of $B_R = A^T A$. Moreover, the ith eigenvalue λ_i of B_R is the ith singular value of A squared: $\lambda_i = \sigma_i^2$.

Similarly we can derive

$$B_L U = A A^T U = (U S V^T)(V S U^T) U = U S^2,$$

and hence the left singular vectors of A are the eigenvectors of $B_L = A A^T$ and the eigenvalues of B_L are the squared singular values of A.

Eigendecomposition. In general, the eigenvectors provide a basis for a matrix $B \in \mathbb{R}^{n \times n}$ in the same way that the right V or left singular vectors U provide a basis for matrix $A \in \mathbb{R}^{n \times d}$. In fact, it is again a very special basis, and is unique up to the multiplicity of eigenvalues. This implies that all eigenvectors are orthogonal to each other.

Let $V = [v_1, v_2, \ldots, v_n]$ be the eigenvectors of the matrix $B \in \mathbb{R}^{n \times n}$, as columns in the matrix V. Also let $L = \mathsf{diag}(\lambda_1, \lambda_2, \ldots, \lambda_d)$ be the eigenvalues of B stored on the diagonal of matrix L. Then we can decompose B as

$$B = V L V^{-1}.$$

[1] Technically, $S \in \mathbb{R}^{n \times d}$. To make this simple argument work, lets first assume w.l.o.g. (without loss of generality) that $d \leq n$. Then the bottom $n - d$ rows of S are all zeros, which mean the right $n - d$ rows of U do not matter. So we can ignore both these $n - d$ rows and columns. Then S is square. This makes U no longer orthogonal, so $U^T U$ is then a projection, not identity; but it turns out this is a project to the span of A, so the argument still works.

copyright: Jeff M. Phillips

Note that the inverse of L is $L^{-1} = \mathsf{diag}(1/\lambda_1, 1/\lambda_2, \ldots, 1/\lambda_n)$. Hence we can write

$$B^{-1} = VL^{-1}V^{-1}.$$

When B is positive definite, it has n positive eigenvectors and eigenvalues; hence V is orthogonal, so $V^{-1} = V^T$. Thus in this situation, given the eigendecomposition, we now have a way to compute the inverse

$$B^{-1} = VL^{-1}V^T,$$

which was required in our almost closed-form solution for linear regression. Now we just need to compute the eigendecomposition, which we will discuss next.

7.4 The Power Method

The *power method* refers to what is probably the simplest algorithm to compute the first eigenvector and value of a matrix. By factoring out the effect of the first eigenvector, we can then recursively repeat the process on the remainder until we have found all eigenvectors and values. Moreover, this implies we can also reconstruct the singular value decomposition as well.

We will consider $B \in \mathbb{R}^{n \times n}$, a positive semidefinite matrix: $B = A^T A$.

Algorithm 7.4.1 PowerMethod(B, q)

initialize $u^{(0)}$ as a random unit vector.
for $i = 1$ **to** q **do**
  $u^{(i)} := Bu^{(i-1)}$
return $v := u^{(q)}/\|u^{(q)}\|$

We can unroll the for loop to reveal another interpretation. We can directly set $v^{(q)} = B^q v^{(0)}$, so all iterations are incorporated into one matrix-vector multiplication. Recall that $B^q = B \cdot B \cdot B \cdot \ldots \cdot B$, for q times. However, these q matrix multiplications are much more expensive then q matrix-vector multiplications.

Alternatively we are provided only the matrix A (where $B = A^T A$) then we can run the algorithm without explicitly constructing B (since for instance if $d > n$ and $A \in \mathbb{R}^{n \times d}$, then the size of B (d^2) may be much larger than A (nd)). Then we simply replace the inside of the for-loop with

$$u^{(i)} := A^T(Au^{(i-1)})$$

where we first multiply $\tilde{u} = Au^{(i-1)}$ and then complete $u^{(i)} = A^T\tilde{u}$.

Recovering all eigenvalues. The output of PowerMethod$(B = A^T A, q)$ is a single unit vector v, which we will argue is arbitrarily close to the first eigenvector v_1. Clearly we can recover the first eigenvalue as $\lambda_1 = \|Bv_1\|$. Since we know the eigenvectors form a basis for B, they are orthogonal. Hence, after we have constructed the first eigenvector v_1, we can factor it out from B as follows:

$$A_1 := A - Av_1v_1^T$$
$$B_1 := A_1^T A_1$$

Then we run PowerMethod$(B_1 = A_1^T A_1, q)$ to recover v_2, and λ_2; factor them out of B_1 to obtain B_2, and iterate.

Geometry of Why the Power Method Works?

To understand why the power method works, assume we know the eigenvectors $v_1, v_2, \ldots, v_n$ and eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of $B \in \mathbb{R}^{n \times n}$.
Since the eigenvectors form a basis for B, and assuming it is full rank, then also for all of $\mathbb{R}^n$ (if not, then it does not have n eigenvalues, and we can fill out the rest of the basis of $\mathbb{R}^n$ arbitrarily). Hence, for any vector, including the initialization random vector $u^{(0)}$ can be written as

$$u^{(0)} = \sum_{j=1}^{n} \alpha_j v_j.$$

Recall that $\alpha_j = \langle u^{(0)}, v_j \rangle$, and since it is random, it is possible to claim that with probability at least $1/2$ that for any α_j we have that $|\alpha_j| \geq \frac{1}{2}\sqrt{n}$[a]. We will now assume that this holds for $j = 1$, so $\alpha_1 > 1/2\sqrt{n}$.
Next since we can interpret that algorithm as $v = B^q u^{(0)}$, then lets analyze B^q. If B has jth eigenvector v_j and eigenvalue λ_j, that is, $Bv_j = \lambda_j v_j$, then B^q has jth eigenvalue λ_j^q since

$$B^q v_j = B \cdot B \cdot \ldots \cdot B v_j = B^{q-1}(v_j \lambda_j) = B^{q-2}(v_j \lambda_j)\lambda_j = v_j \lambda_j^q.$$

This holds for each eigenvalue of B^q. Hence we can rewrite output by summing over the terms in the eigenbasis as

$$v = \frac{\sum_{j=1}^{n} \alpha_j \lambda_j^q v_j}{\sqrt{\sum_{j=1}^{n} (\alpha_j \lambda_j^q)^2}}.$$

Finally, we would like to show our output v is close to the first eigenvector v_1. We can measure closeness with the dot product (actually we will need to use its absolute value since we might find something close to $-v_1$).

$$\begin{aligned}
|\langle B^q u^{(0)}, v_1 \rangle| &= \frac{\alpha_1 \lambda_1^q}{\sqrt{\sum_{j=1}^{n} (\alpha_j \lambda_j^q)^2}} \\
&\geq \frac{\alpha_1 \lambda_1^q}{\sqrt{\alpha_1^2 \lambda_1^{2q} + n\lambda_2^{2t}}} \geq \frac{\alpha_1 \lambda_1^q}{\alpha_1 \lambda_1^q + \lambda_2^q \sqrt{n}} = 1 - \frac{\lambda_2^q \sqrt{n}}{\alpha_1 \lambda_1^q + \lambda_2^q \sqrt{n}} \\
&\geq 1 - 2\sqrt{n} \left(\frac{\lambda_2}{\lambda_1}\right)^q.
\end{aligned}$$

The first inequality holds because $\lambda_1 \geq \lambda_2 \geq \lambda_j$ for all $j > 2$. The third inequality (going to third line) holds by dropping the $\lambda_2^q \sqrt{n}$ term in the denominator, and since $\alpha_1 > 1/2\sqrt{n}$.
Thus if there is "gap" between the first two eigenvalues (λ_1/λ_2 is large), then this algorithm converges quickly to where $|\langle v, v_1 \rangle| = 1$.

[a] Since $u^{(0)}$ is a unit vector, its norm is 1, and because $\{v_1, \ldots, v_n\}$ is a basis, then $1 = \|u^{(0)}\|^2 = \sum_{j=1}^{n} \alpha_j^2$. Since it is random, then $\mathbf{E}[\alpha_j^2] = 1/n$ for each j. Applying a concentration of measure (almost a Markov Inequality, but need to be more careful), we can argue that with probability $1/2$ any $\alpha_j^2 > (1/4) \cdot (1/n)$, and hence $\alpha_j > (1/2) \cdot (1/\sqrt{n})$.

copyright: Jeff M. Phillips

7.5 Principal Component Analysis

Recall that the original goal of this topic was to find the k-dimensional subspace F to minimize

$$\|A - \pi_F(A)\|_F^2 = \sum_{a_i \in A} \|a_i - \pi_F(a_i)\|^2.$$

We have not actually solved this yet. The top k right singular values V_k of A only provided this bound assuming that F contains the origin: $(0, 0, \ldots, 0)$. However, this might not be the case!

Principal Component Analysis (PCA) is an extension of the SVD when we do not restrict that the subspace V_k must go through the origin. It turns out, like with simple linear regression, that the optimal F must go through the mean of all of the data. So we can still use the SVD, after a simple preprocessing step called centering to shift the data matrix so its mean is exactly at the origin.

Specifically, *centering* is adjusting the original input data matrix $A \in \mathbb{R}^{n \times d}$ so that each column (each dimension) has an average value of 0. This is easier than it seems. Define $\bar{a}_j = \frac{1}{n}\sum_{i=1}^n A_{i,j}$ (the average of each column j). Then set each $\tilde{A}_{i,j} = A_{i,j} - \bar{a}_j$ to represent the entry in the ith row and jth column of centered matrix $\tilde{A}$.

There is a *centering matrix* $C_n = I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^T$ where I_n is the $n \times n$ identity matrix, $\mathbf{1}$ is the all-ones column vector (of length n) and thus $\mathbf{1}\mathbf{1}^T$ is the all-ones $n \times n$ matrix. Then we can also just write $\tilde{A} = C_n A$.

Now to perform PCA on a data set A, we compute $[U, S, V] = \texttt{svd}(C_n A) = \texttt{svd}(\tilde{A})$.

Then the resulting singular values $\texttt{diag}(S) = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ are known as the *principal values*, and the top k right singular vectors $V_k = [v_1\ v_2\ \ldots\ v_k]$ are known as the top-k *principal directions*.

This often gives a better fitting to the data than just SVD. The SVD finds the best rank-k approximation of A, which is the best k-dimensional subspace (up to Frobenius and spectral norms) **which passes through the origin**. If all of the data is far from the origin, this can essentially "waste" a dimension to pass through the origin. However, we also need to store the shift from the origin, a vector $\tilde{c} = (\tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_d) \in \mathbb{R}^d$.

7.6 Multidimensional Scaling

Dimensionality reduction is an abstract problem with input of a high-dimensional data set $P \subset \mathbb{R}^d$ and a goal of finding a corresponding lower dimensional data set $Q \subset \mathbb{R}^k$, where $k << d$, and properties of P are preserved in Q. Both low-rank approximations through direct SVD and through PCA are examples of this: $Q = \pi_{V_k}(P)$. However, these techniques require an explicit representation of P to start with. In some cases, we are only presented P more abstractly. There two common situations:

- We are provided a set of n objects X, and a bivariate function $\texttt{d} : X \times X \to \mathbb{R}$ that returns a distance between them. For instance, we can put two cities into an airline website, and it may return a dollar amount for the cheapest flight between those two cities. This dollar amount is our "distance."
- We are simply provided a matrix $D \in \mathbb{R}^{n \times n}$, where each entry $D_{i,j}$ is the distance between the ith and jth point. In the first scenario, we can calculate such a matrix D.

Multi-Dimensional Scaling (MDS) has the goal of taking such a distance matrix D for n points and giving low-dimensional (typically) Euclidean coordinates to these points so that the embedded points have similar spatial relations to that described in D. If we had some original data set A which resulted in D, we could just apply PCA to find the embedding. It is important to note, in the setting of MDS we are typically just given D, and *not* the original data A. However, as we will show next, we can derive a matrix that will act like AA^T using only D.

A *similarity matrix* M is an $n \times n$ matrix where entry $M_{i,j}$ is the similarity between the ith and the jth data point. The similarity often associated with Euclidean distance $\|a_i - a_j\|$ is the standard inner (or dot

product) $\langle a_i, a_j \rangle$. We can write

$$\|a_i - a_j\|^2 = \|a_i\|^2 + \|a_j\|^2 - 2\langle a_i, a_j \rangle,$$

and hence

$$\langle a_i, a_j \rangle = \frac{1}{2}\left(\|a_i\|^2 + \|a_j\|^2 - \|a_i - a_j\|^2\right). \tag{7.1}$$

Next we observe that for the $n \times n$ matrix AA^T the entry $[AA^T]_{i,j} = \langle a_i, a_j \rangle$. So it seems hopeful we can derive AA^T from D using equation (7.1). That is we can set $\|a_i - a_j\|^2 = D_{i,j}^2$. However, we need also need values for $\|a_i\|^2$ and $\|a_j\|^2$.

Since the embedding has an arbitrary shift to it (if we add a shift vector s to *all* embedding points, then no distances change), then we can arbitrarily choose a_1 to be at the origin. Then $\|a_1\|^2 = 0$ and $\|a_j\|^2 = \|a_1 - a_j\|^2 = D_{1,j}^2$. Using this assumption and equation (7.1), we can then derive the similarity matrix AA^T. Then we can run the eigen-decomposition on AA^T and use the coordinates of each point along the first k eigenvectors to get an embedding. This is known as *classical MDS*.

It is often used for k as 2 or 3 so the data can be easily visualized.

There are several other forms that try to preserve the distance more directly, where as this approach is essentially just minimizing the squared residuals of the projection from some unknown original (high-dimensional embedding). One can see that we recover the distances with no error if we use all n eigenvectors – if they exist. However, as mentioned, there may be less than n eigenvectors, or they may be associated with complex eigenvalues. So if our goal is an embedding into $k = 3$ or $k = 10$, there is no guarantee that this will work, or even what guarantees this will have. But MDS is used a lot nonetheless.

 copyright: Jeff M. Phillips

Exercises

We will use a dataset, here: `http://www.cs.utah.edu/~jeffp/teaching/FoDA/A.csv`

Q7.1: Read data set `A.csv` as a matrix $A \in \mathbb{R}^{30\times 6}$. Compute the SVD of A and report

1. the third right singular vector,
2. the second singular value, and
3. the fourth left singular vector.
4. What is the rank of A?

Compute A_k for $k = 2$.

1. What is $\|A - A_k\|_F^2$?
2. What is $\|A - A_k\|_2^2$?

Center A. Run PCA to find the best 2-dimensional subspace F to minimize $\|A - \pi_F(A)\|_F^2$. Report

1. $\|A - \pi_F(A)\|_F^2$ and
2. $\|A - \pi_F(A)\|_2^2$.

Q7.2: Consider another matrix $A \in \mathbb{R}^{8\times 4}$ with squared singular values $\sigma_1^2 = 10$, $\sigma_2^2 = 5$, $\sigma_3^2 = 2$, and $\sigma_4^2 = 1$.

1. What is the rank of A?
2. What is $\|A - A_2\|_F^2$, where A_2 is the best rank-2 approximation of A.
3. What is $\|A - A_2\|_2^2$, where A_2 is the best rank-2 approximation of A.
4. What is $\|A\|_2^2$?
5. What is $\|A\|_F^2$?

Let v_1, v_2, v_3, v_4 be the right singular vectors of A.

1. What is $\|Av_2\|^2$?
2. What is $\langle v_1, v_3\rangle$?
3. What is $\|v_4\|$?

Let $a_1 \in \mathbb{R}^4$ be the first row of A.

1. Write a_1 in the basis defined by the right singular vectors of A.

Q7.3: Consider two matrices A_1 and A_2 both in $\mathbb{R}^{10\times 3}$. A_1 has singular values $\sigma_1 = 20$, $\sigma_2 = 2$, and $\sigma_3 = 1.5$. A_2 has singular values $\sigma_1 = 8$, $\sigma_2 = 4$, and $\sigma_3 = 0.001$.

1. For which matrix will the power method converge faster to the top eigenvector of $A_1^T A_1$ (or $A_2^T A_2$, respectively), and why?

Given the eigenvectors v_1, v_2, v_3 of $A^T A$. Explain step by step how to recover the following. Specifically, you should write the answers as linear algebraic expressions in terms of v_1, v_2, v_3, and A; it can involve taking norms, matrix multiply, addition, subtraction, but not something more complex like SVD.

copyright: Jeff M. Phillips

2. the second singular value of A
3. the first right singular vector of A
4. the third left singular vector of A

copyright: Jeff M. Phillips

8 Clustering

This topic will focus on automatically grouping data points into subsets of similar points. There are numerous ways to define this problem, and most of them are quite messy. And many techniques for clustering actually lack a mathematical formulation. We will focus on what is probably the cleanest and most used formulation: k-means clustering. But, for background, we will begin with a mathematical detour in Voronoi diagrams.

8.1 Voronoi Diagrams

Consider a set $S = \{s_1, s_2, \ldots, s_k\} \subset \mathbb{R}^d$ of sites. We would like to understand how these points carve up the space $\mathbb{R}^d$.

We can think of this more formally as the *post office problem*. Let these sites define the locations of a post office. For all points in $\mathbb{R}^d$ (e.g., a point on the map for points in $\mathbb{R}^2$), we would like to assign it to the closest post office. For a fixed point, we can just check the distance to each post office:

$$\phi_S(x) = \arg\min_{s_i \in S} \|x - s_i\|.$$

However, this may be slow (naively take $O(k)$ time for each point x), and does not provide a general representation or understanding for all points. The "correct" solution to this problem is the Voronoi diagram.

The Voronoi diagram decomposes $\mathbb{R}^d$ into k regions (a *Voronoi cell*), one for each site. The region for site s_i is defined.

$$R_i = \{x \in \mathbb{R}^d \mid \phi_S(x) = s_i\}$$

If we have these regions nicely defined, this solves the post office problem. For any point x, we just need to determine which region it lies in (for instance in $\mathbb{R}^2$, once we have defined these regions, through an extension of binary search, we can locate the region containing any $x \in \mathbb{R}^2$ in only $O(\log k)$ time). But what do these regions look like, and what properties do they have.

Voronoi edges and vertices. We will start our discussion in $\mathbb{R}^2$. Further, we will assume that the sites S are in *general position*: in this setting, it means that no set of three points lie on a common line, and that no set of four points lie on a common circle.

The boundary between two regions R_i and R_j, called a *Voronoi edge*, is a line or line segment. This edge $e_{i,j}$ is defined

$$e_{i,j} = \{x \in \mathbb{R}^2 \mid \|x - s_i\| = \|x - s_j\| \leq \|x - s_\ell\| \text{ for all } \ell \neq i, j\}$$

as the set of all points equal distance to s_i and s_j, and not closer to any other point s_ℓ.

Why is this set a line segment? If we only have two points in S, then it is the bisector between them. Draw a circle centered at any point x on this bisector, and if it intersects one of s_i or s_j, it will also intersect the other. This is true since we can decompose the squared distance from x to s_i along orthogonal components: along the edge, and perpendicular to the edge from s_i to $\pi_{e_{i,j}}(s_i)$.

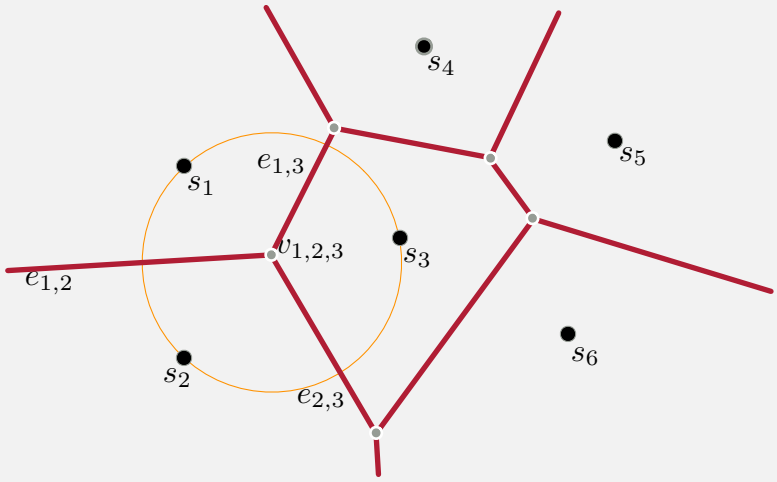
Similarly, a Voronoi vertex $v_{i,j,\ell}$ is a point where three sites s_i, s_j, and s_ℓ are all equidistance, and no other points are closer:

$$v_{i,j,k} = \{x \in \mathbb{R}^2 \mid \|x - s_i\| = \|x - s_j\| = \|x = s_\ell\| \leq \|x - s_k\| \text{ for all } k \neq i, j, \ell\}.$$

This vertex is the intersection (and end point) of three Voronoi edges $e_{i,i}$, $e_{i,\ell}$, and $e_{j,\ell}$. Think of sliding a point x along an edge $e_{i,j}$ and maintaining the circle centered at x and touching s_i and s_j. When this circle grows to where it also touches s_ℓ, then $e_{i,j}$ stops.

Example: Voronoi Diagram

See the following example with $k = 6$ sites in $\mathbb{R}^2$. Notice the following properties: edges may be unbounded, and the same with regions. The circle centered at $v_{1,2,3}$ passes through s_1, s_2, and s_3. Also, Voronoi cell R_3 has $5 = k - 1$ vertices and edges.



Size complexity. So how complicated can these Voronoi diagrams get? A single Voronoi cell can have $k-1$ vertices and edges. So can the entire complex be of size $O(k^2)$ (each of k regions requiring complexity $O(k)$)? No. The Voronoi vertices and edges describe a planar graph. Some cool results from graph theory says that planar graphs have asymptotically the same number of edges, faces, and vertices. Euler's Formula for a planar graph with n vertices, m edges, and k faces is that $k + n - m = 2$. And Kuratowski's criteria says for $n \geq 3$, then $m \leq 3n - 6$. Hence, $k \leq 2n - 4$ for $n \geq 3$. The duality construction to Delauney triangulations (discussed below) will complete the argument. Since there are k faces (the k Voronoi cells, one for each site), then there are also $O(k)$ edges and $O(k)$ vertices. In particular, there will be precisely $2k - 5$ vertices and $3k - 6$ edges.

However, this does not hold in $\mathbb{R}^3$. In particular, for $\mathbb{R}^3$ and $\mathbb{R}^4$, the complexity (number of cells, vertices, edges, faces, etc) is $O(k^2)$. This means, there could be roughly as many edges as their are pairs of vertices!

But it can get much worse. In $\mathbb{R}^d$ (for general d) then the complexity is $O(k^{\lceil d/2 \rceil})$. This is a lot. Hence, this structure is impractical to construct in high dimensions.

: *The **curse** of dimensionality! ooooh*

Moreover, since this structure is explicitly tied to the post office problem, and the nearest neighbor function ϕ_S, it indicates that (a) in $\mathbb{R}^2$ this function is nicely behaved, but (b) in high dimensions, it is quite complicated.

8.1.1 Delaunay Triangulation

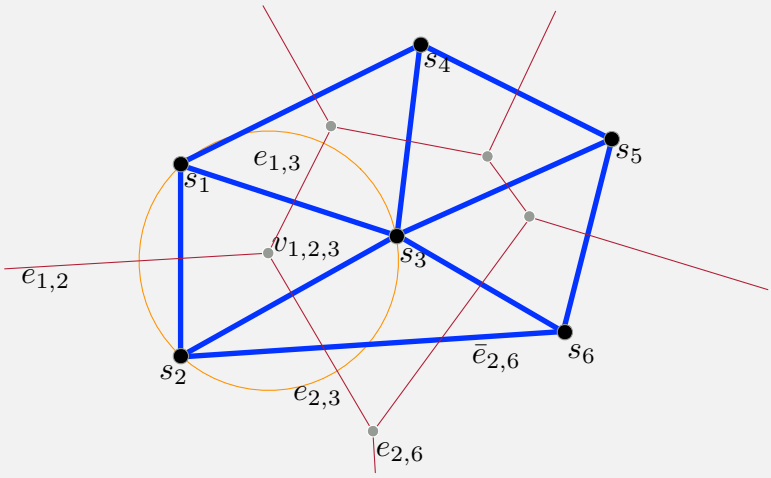
A fascinating aspect of the Voronoi diagram is that it can be converted into a very special graph where the sites S are vertices, one called the *Delaunay triangulation*. This is the *dual* of the Voronoi diagram.

- Each face R_i of the Voronoi diagram maps to a vertex s_i in the Delaunay triangulation.
- Each vertex $v_{i,j,\ell}$ in the Voronoi diagram maps to a triangular face $f_{i,j,\ell}$ in the Delaunay triangulation.
- Each edge $e_{i,j}$ in the Voronoi diagram maps to an edge $\bar{e}_{i,j}$ in the Delaunay triangulation.

copyright: Jeff M. Phillips

Example: Delaunay Triangulation

See the following example with 6 sites in $\mathbb{R}^2$. Notice that every edge, face, and vertex in the Delaunay triangulation corresponds to a edge, vertex, and face in the Voronoi diagram. Interestingly, the associated edges may not intersection; see $e_{2,6}$ and $\bar{e}_{2,6}$.



Because of the duality between the Voronoi diagram and the Delaunay triangulation, their complexities are the same. That means the Voronoi diagram is of size $O(k)$ for k sites in $\mathbb{R}^2$, but more generally is of size $O(k^{\lceil d/2 \rceil})$ in $\mathbb{R}^d$.

The existence of of the Delaunay triangulation shows that there always exist a triangulation: A graph with vertices of a given set of points $S \subset \mathbb{R}^2$ so that all edges are straightline segments between the vertices, and each face is a triangle. In fact, there are many possible triangulations: one can always simply construct some triangulation greedily, draw any possible edges that does not cross other edges until no more can be drawn.

The Delaunay triangulation, however, is quite special. This is the triangulation that maximizes the smallest angle over all triangles; for meshing applications in graphics and simulation, skinny triangles are very hard to deal with, so this is very useful.

In circle property. Another cool way to define the Delaunay triangulation is through the *in circle* property. For any three points, the smallest enclosing ball either has all three points on the boundary, or has two points on the boundary and they are antipodal to each other. Any circle with two points antipodal on the boundary s_i and s_j, and contains no other points, then the edge $e_{i,j}$ is in the Delaunay triangulation. This is a subset of the Delaunay triangulation called the *Gabriel graph*.

Any circle with three points on its boundary s_i, s_j, and s_ℓ, and no points in its interior, then the face $f_{i,j,\ell}$ is in the Delaunay triangulation, as well as its three edges $e_{i,j}$, $e_{i,\ell}$ and $e_{j,\ell}$. But does not imply those edges satisfy the Gabriel property.

For instance, on a quick inspection, (in the example above) it may not be clear if edge $e_{3,5}$ or $e_{4,6}$ should be in the Delaunay triangulation. Clearly it can not be both since they cross. But the ball with boundary through s_3, s_4, and s_6 would contain s_5, so the face $f_{3,4,6}$ cannot be in the Delaunay triangulation. On the other hand the ball with boundary through s_3, s_6, and s_5 does not contain s_4 or any other points in S, so the face $f_{3,5,6}$ is in the Delaunay triangulation.

 copyright: Jeff M. Phillips

8.1.2 Connection to Clustering

So what is the connection to clustering? Given a large set $X \subset \mathbb{R}^d$ of size n, we would like to find a set of k sites S (post office locations) so that each point $x \in X$ is near some post office. This is a proxy problem. So given a set of sites S, determining for each $x \in X$ which site is closest is exactly determined by the Voronoi diagram.

8.2 k-Means Clustering

Probably the most famous clustering formulation is k-means. The term "k-means" refers to a problem formulation, *not an algorithm*. There are many algorithms with aim of solving the k-means problem formulation, exactly or approximately. We will mainly focus on the most common: Lloyd's algorithm. Unfortunately, it is common in the literature to see "the k-means algorithm," this typically should be replaced with Lloyd's algorithm.

k-Means is in the family of *assignment-based clustering*. Each cluster is represented by a single point, to which all other points in the cluster are "assigned." The input is a data set X, and the output is a set of centers $S = \{s_1, s_2, \ldots, s_k\}$. This implicitly defines a set of clusters where $\phi_S(x) = \arg\min_{s \in S} \|x - s\|$ (the same as in the post office problem). Then the *k-means clustering problem* is to find the set S of k clusters to minimize

$$\mathsf{cost}(X, S) = \sum_{x \in X} \|\phi_S(x) - x\|^2.$$

So we want every point assigned to the closest site, and want to minimize the sum of the squared distance of all such assignments.

8.3 Lloyd's Algorithm

When people think of the k-means problem, they usually think of the following algorithm. It is usually attributed to Stuart P. Lloyd from a document in 1957, although it was not published until 1982.[1]

Algorithm 8.3.1 Lloyd's Algorithm for k-Means Clustering

Choose k points $S \subset X$ *arbitrarily?*
repeat
  for all $x \in X$: assign x to X_i for $s_i = \phi_S(x)$ *the closest site $s \in S$ to x*
  for all $s_i \in S$: set $s_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$ *the average of $X_i = \{x \in X \mid \phi_S(x) = s_i\}$*
until (the set S is unchanged)

Convergence. If the main loop has R rounds, then this takes roughly Rnk steps (and can be made closer to $Rn \log k$ with faster nearest neighbor search in some cases). But how large is R; that is, how many iterations do we need?

First, we can argue that the number of steps is finite. This is true since the cost function $\mathsf{cost}(X, S)$ will always decrease. To see this, writing it as a sum over S.

$$\mathsf{cost}(X, S) = \sum_{x \in X} \|\phi_S(x) - x\|^2 = \sum_{s_i \in S} \sum_{x \in X_i} \|s_i - x\|^2.$$

[1] Apparently, the IBM 650 computer Lloyd was using in 1957 did not have enough computational power to run the (very simple) experiments he had planned. This was replaced by the IBM 701, but it did not have quite the same "quantization" functionality as the IBM 650, and the work was forgotten. Lloyd was also worried about some issues regarding the k-means problem not having a unique minimum.

copyright: Jeff M. Phillips

Then in each step of the **repeat-until** loop, thus must decrease. The first step holds since it moves each $x \in X$ to a subset X_i with the corresponding center s_i closer to (or the same distance to) x than before. So for each x the term $\|x - s_i\|$ is reduced (or the same). The second step holds since for each inner sum $\sum_{x \in X_i} \|s_i - x\|^2$, the single point s_i which minimizes this cost is precisely the average of X_i. So reassigning s_i as described also decreases the cost (or keeps it the same).

Importantly, if the cost decreases each step, then it cannot have the same set of centers S on two different steps, since that would imply the assignment sets $\{X_i\}$ would also be the same. Thus, in order for this to happen, the cost would need to decrease after the first occurrence, and then increase to obtain the second occurrence, which is not possible.

Since, there are finite ways each set of points can be assigned to different clusters, then, the algorithm terminates in a finite number of steps.

... but in practice usually we may run for $R = 10$, or maybe $R = 20$ steps. Or check if the change in cost function is below some sufficiently small threshold.

On clusterability: When data is easily clusterable, most clustering algorithms work quickly and well. When is not easily clusterable, then no algorithm will find good clusters.

Sometimes there is a good k-means clustering, but it is not found by Lloyd's algorithm. Then we can choose new centers again (with randomness), and try again.

Initialization. The initial paper by Lloyd advocates to choose the initial partition of X into disjoint subsets $X_1, X_2, \ldots, X_k$ *arbitrarily*. However, some choices will not be very good. For instance, if we randomly place each $x \in X$ into some x_i, then (by the central limit theorem) we expect all $s_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$ to all be close to the mean of the full data set $\frac{1}{|X|} \sum_{x \in X} x$.

A bit safer way to initialize the data is to choose a set $S \subset X$ at random. Since each s_i is chosen separately (not as an average of data points), there is no centering phenomenon. However, even with this initialization, we may run Lloyd's algorithm to completion, and find a sub-optimal solution (*a local minimum!*). Thus, it is usually safer to randomly re-initialize the algorithm several times (say 3-5 times) and rerun Lloyd's algorithm for each. the probability all random restarts results in a local minimum is more rare.

A more principled way to choose an initial set S is to use an algorithm like Gonzalez (for k steps, iteratively chose the point $x \in X$ furthest from all sites chosen so far), or k-means++ (for k steps, iteratively choose a point at random proportional to the squared distance to its nearest already chosen site). These are beyond the scope of this class, but offer much stronger error guarantees of various forms. But, for k-means++, it is still suggested to try random restarts.

```
from sklearn.cluster import KMeans
import numpy as np
X = np.array([[2, 3], [2, 4], [2, 2],[7, 1], [7, 2], [7, 3]])
kmeans = KMeans(n_clusters=2).fit(X)

print kmeans.labels_
# [1 1 1 0 0 0]

print kmeans.cluster_centers_
#[[ 7.  2.]
# [ 2.  3.]]
```

Number of clusters So what is the right value of k? Like with PCA, there is no perfect answer towards choosing how many dimensions the subspace should be. When k is not given to you, typically, you would run with many different values of k. Then create a plot of $\mathsf{cost}(S, X)$ as a function of k. This cost will

always decrease with larger k; but of course $k = n$ is of no use. At some point, the cost will not decrease much between values (this implies that probably two centers are used in the same grouping of data, so the squared distance to either is similar). Then this is a good place to choose k.

8.3.1 Extensions to the k-Means Problem and Lloyd's Algorithm

Like many algorithms in this class, Lloyd's depends on the use of a SSE cost function. In particular, it works because for $X \in \mathbb{R}^d$, then

$$\frac{1}{|X|}\sum_{x\in X} x = \text{average}(X) = \arg\min_{s\in\mathbb{R}^d} \sum_{x\in X} \|s - x\|^2.$$

There are not similar properties for other costs functions, or when X is not in $\mathbb{R}^d$. For instance, one may want to solve the k-medians problem where one just minimizes the sum of (non-squared) distances. In particular, this has no closed form solution for $X \in \mathbb{R}^d$ for $d > 1$.

An alternative to the averaging step is to choose

$$s_i = \arg\min_{s\in X_i} \mathtt{d}(x, s)$$

where $\mathtt{d}(x, s)$ is an arbitrary measure (like non-squared distance) between x and s. That is, we choose an s from the set X_i. This is particularly useful when X is in a non-Euclidean metric space where averaging may not be well-defined. For the specific case where $\mathtt{d}(x, s) = \|x - s\|$ (for the k-median problem), then this variant of the algorithm is called *k-mediods*.

Soft clustering. Sometimes it is not desirable to assign each point to exactly one cluster. Instead, we may split a point between one or more clusters, assigning a fractional value to each. This is known as *soft clustering* whereas the original formulation is known as *hard clustering*.

There are many ways to achieve a soft clustering. For instance, consider the following Voronoi diagram-based approach called based on natural neighbor interpolation (NNI). Let $V(S)$ be the Voronoi diagram of the sites S (which decomposes $\mathbb{R}^d$). Then construct $V(S \cup x)$ for a particular data point x; the Voronoi diagram of the sites S with the addition of one point x. For the region R_x defined by the point x in $V(S\cup x)$, overlay it on the original Voronoi diagram $V(S)$. This region R_x will overlap with regions R_i in the original Voronoi diagram; compute the volume v_i for the overlap with each such region. Then the fractional weight for x into each site s_i is defined $w_i(x) = v_i / \sum_{i=1}^n v_i$.

We can plug any such step into Lloyd's algorithm, and then recalculate s_i as the weighted average of all points partially assigned to the ith cluster.

8.4 Mixture of Gaussians

The k-means formulation tends to define clusters of roughly equal size. The squared cost discourages points far from any center. It also, does not adapt much to the density of individual centers.

An extension is to fit each cluster X_i with a Gaussian distribution $\mathcal{G}(\mu_i, \Sigma_i)$, defined by a mean μ_i and a covariance matrix Σ_i. Recall that the pdf of a d-dimensional Gaussian distribution is defined

$$f_{\mu,\Sigma}(x) = \frac{1}{(2\pi)^{d/2}} \frac{1}{\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

where $|\Sigma|$ is the determinant of Σ. For instance, for $d = 2$, and the standard deviation in the x-direction of X is σ_x, and in the y-direction is σ_y, and their correlation is ρ, then

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}.$$

Now the goal is, given a parameter k, find a set of k pdfs $F = \{f_1, f_2, \ldots, f_k\}$ where $f_i = f_{\mu_i, \Sigma_i}$ to maximize

$$\prod_{x \in X} \max_{f_i \in F} f_i(x),$$

or equivalently to minimize

$$\sum_{x \in X} \min_{f_i \in F} -\log(f_i(x)).$$

For the special case where when we restrict that $\Sigma_i = I$ (the identity matrix) for each mixture, then one can check that the second formulation (the log-likelihood version) is equivalent to the k-means problem.

This hints that we can adapt Lloyds algorithm towards this problem as well. To replace the first step of the inner loop, we assign each $x \in X$ to the Gaussian which maximizes $f_i(x)$:

$$\text{for all } x \in X\text{: assign } x \text{ to } X_i \text{ so } i = \arg\max_{i \in 1 \ldots k} f_i(x).$$

But for the second step, we need to replace a simple average with an estimation of the best fitting Gaussian to a data set X_i. This is also simple. First, calculate the mean as $\mu_i = \frac{1}{|X_i|} \sum_{x \in X_i} x$. Then calculate the covariance matrix Σ_i of X_i as the sum of outer products

$$\Sigma_i = \sum_{x \in X_i} (x - \mu_i)(x - \mu_i)^T.$$

This can be interpreted as calling PCA. Calculating μ_i, and subtracting from each $x \in X_i$ is the centering step. Letting $\bar{X}_i = \{x - \mu_i \mid x \in X_i\}$, then $\Sigma_i = V S^2 V^T$ where $[U, S, V] = \mathsf{svd}(\bar{X}_i)$.

8.4.1 Expectation-Maximization

The standard way to fit a mixture of Gaussians actually uses a soft-clustering.

Each point $x \in X$ is given a weight $w_i = f_i(x) / \sum_i f_i(x)$ for its assignment to each cluster. Then the mean and covariance matrix is estimated using weight averages.

Algorithm 8.4.1 EM Algorithm for Mixture of Gaussians

Choose k points $S \subset X$ — *arbitrarily?*
for all $x \in X$: set $w_i(x)$ for $s_i = \phi_S(x)$, and $w_i(x) = 0$ otherwise
repeat
  for $i \in [1 \ldots k]$ **do**
    Calculate $W_i = \sum_{x \in X} w_i(x)$ — *the total weight for cluster i*
    Set $\mu_i = \frac{1}{W_i} \sum_{x \in X} w_i(x) x$ — *the weighted average*
    Set $\Sigma_i = \frac{1}{W_i} \sum_{x \in X} w_i (x - \mu_i)(x - \mu_i)^T$ — *the weighted covariance*
  for $x \in X$ **do**
    for all $i \in [1 \ldots k]$: set $w_i(x) = f_i(x) / \sum_i f_i(x)$ — *partial assignments using $f_i = f_{\mu_i, \Sigma_i}$*
until ($\sum_{x \in X} \sum_{i=1}^{k} -\log(w_i(x) \cdot f_i(x))$ has small change)

This procedure is the classic example of a framework called *expectation-maximization*. This is an alternate optimization procedure, which alternates between maximizing the probability of some model (the partial assignment step) and calculating the most likely model using expectation (the average, covariance estimating step).

But this is a much more general framework. It is particularly useful in situations (like this one) where the true optimization criteria is messy and complex, often non-convex; but it can be broken into two or more steps where each step can be solved with a (near) closed form. Or if there is no closed form, but each part is individually convex, the gradient descent can be invoked.

 copyright: Jeff M. Phillips

8.5 Mean Shift Clustering

Now for something completely different. Clustering is a very very broad field with no settled upon approach. To demonstrate this, we will quickly review an algorithm called *mean shift clustering*. This algorithm shifts each data point individually to its weighted center of mass. It terminates when all points converge to isolated sets.

First begin with a bivariate kernel function $K : X \times X \to \mathbb{R}$ such as the (unnormalized) Gaussian kernel

$$K(x, p) = \exp(-\|x - p\|^2/\sigma^2)$$

for some given bandwidth parameter σ. The weighted center of mass around each point $p \in X$ is then defined as

$$\mu(p) = \frac{\sum_{x \in X} K(x, p)x}{\sum_{x \in X} K(x, p)}.$$

The algorithm just shifts each point to its center of mass: $p \leftarrow \mu(p)$.

Algorithm 8.5.1 Mean Shift

repeat
  for all $p \in X$: calculate $\mu(p) = \frac{\sum_{x \in X} K(x,p)x}{\sum_{x \in X} K(x,p)}$.
  for all $p \in X$: set $p \leftarrow \mu(p)$.
until (the average change $\|p - \mu(p)\|$ is small)

This algorithm does not require a parameter k. However, it has other parameters, most notably the choice of kernel K and its bandwidth σ. With the Gaussian kernel (since it has infinite support, $K(x, p) > 0$ for all x, p), it will only stop when all x are at the same point. Thus the termination condition is also important. Alternatively, a different kernel with bounded support may terminate automatically (without a specific condition); for this reason (and for speed) truncated Gaussians are often used.

This algorithm not only clusters the data, but also is a key technique for *de-noising* data. This is a process that not just removes noise (as often thought of as outliers), but attempts to return point to where they should have been before being perturbed by noise – similar to mapping a point to its cluster center.

Geometry of Kernel Density Estimates

A kernel density estimate is a powerful modeling tool that allows one to take a finite set of points $P \subset \mathbb{R}^d$ and a smoothing operator, a kernel K, and transform them into a continuous function $\text{KDE}_P : \mathbb{R}^d \to \mathbb{R}$.

The most common kernel is the Gaussian kernel

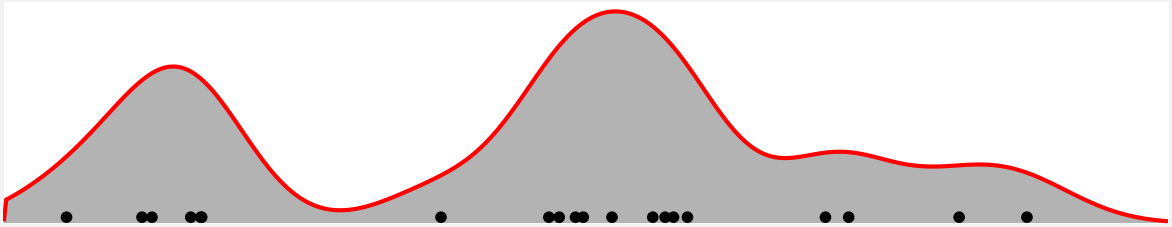
$$K(p, x) = \frac{1}{\sigma^d \sqrt{2\pi^d}} \exp(-\|x - p\|^2/(2\sigma^2)),$$

which is precisely the evaluation of the Gaussian distribution $N_d(x)$ with mean p and standard deviation σ evaluated at x. Thus it can be interpreted as taking the "mass" of a point p and spreading it out according to this probability distribution.

Now, a *kernel density estimate* is defined at a point $x \in \mathbb{R}^d$ as

$$\text{KDE}_P(x) = \frac{1}{|P|} \sum_{p \in P} K(p, x).$$

That is, it gives each point $p \in P$ a mass of $\frac{1}{|P|}$ (each weighted uniformly), then spreads each of these $|P|$ masses out using the Gaussian kernels, and sums them up. While P is discrete in nature, the kernel density estimate KDE_P allows one to interpolate between these points in a smooth natural way.



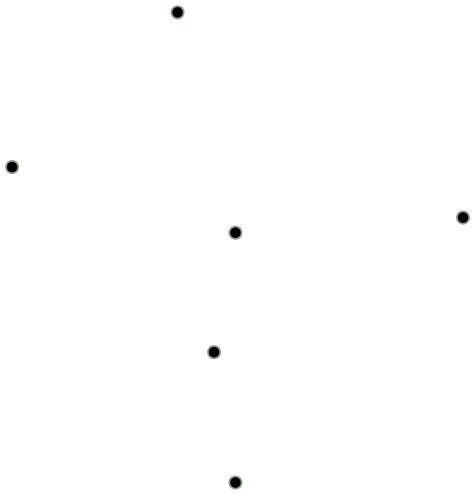
 copyright: Jeff M. Phillips

Exercises

We will use two datasets, here: `http://www.cs.utah.edu/~jeffp/teaching/FoDA/P.csv` and here: `http://www.cs.utah.edu/~jeffp/teaching/FoDA/Q.csv`

Q8.1: Download data sets P and Q. Both have 120 data points, each in 6 dimensions, can be thought of as data matrices in $\mathbb{R}^{120\times 6}$. For each, run some algorithm to construct the k-means clustering of them. Diagnose how many clusters you think each data set should have by finding the solution for k equal to $1, 2, 3, \ldots, 10$.

Q8.2: Draw the Voronoi diagram of the following set of points.



Q8.3: What should you do, if running Lloyd's algorithm for k-means clustering ($k = 2$), and you reach this scenario, where the algorithm terminates? (The black circles are data points and red stars are the centers).

Q8.4: Construct a data set X with 5 points in $\mathbb{R}^2$ and a set S of $k = 3$ sites so that Lloyds algorithm will have converged, but there is another set S', of size $k = 3$, so that $\mathsf{cost}(X, S') < \mathsf{cost}(X, S)$. Explain why S' is better than S, but that Lloyds algorithm will not move from S.

copyright: Jeff M. Phillips

9 Classification

This topic returns to prediction. Unlike linear regression where we were predicting a numeric value, in this case we are predicting a class: winner or loser, yes or no, rich or poor, positive or negative. Ideas from linear regression can be applied here, but we will instead overview a different, but still beautiful family of techniques based on linear classification.

This is perhaps *the* central problem in data analysis. For instance, you may want to predict:

- will a sports team win a game?
- will a politician be elected?
- will someone like a movie?
- will someone click on an ad?
- will I get a job? (If you can build a good classifier, then probably yes!)

Each of these is typically solved by building a general purpose classifier (about sports or movies etc), then applying it to the problem in question.

9.1 Linear Classifiers

Our input here is a point set $X \subset \mathbb{R}^d$, where each element $x_i \in X$ also has an associated label y_i. And $y_i \in \{-1, +1\}$.

Like in regression, our goal is prediction and generalization. We assume each $(x_i, y_i) \sim \mu$; that is, each data point pair, is drawn iid from some fixed but unknown distribution. Then our goal is a function $g : \mathbb{R}^d \to \mathbb{R}$, so that if $y_i = +1$, then $g(x_i) \geq 0$ and if $y_i = -1$, then $g(x_i) \leq 0$.

We will restrict that g is linear. For a data point $x \in \mathbb{R}^d$, written $x = (x^{(1)}, x^{(2)}, \ldots, x^{(d)})$ we enforce that

$$g(x) = \alpha_0 + x^{(1)}\alpha_1 + x^{(2)}\alpha_2 + \ldots + x^{(d)}\alpha_d = \alpha_0 + \sum_{j=1}^{d} x^{(j)}\alpha_j,$$

for some set of scalar parameters $\alpha = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_d)$. Typically, different notation is used: we set $b = \alpha_0$ and $w = (w_1, w_2, \ldots, w_d) = (\alpha_1, \alpha_2, \ldots, \alpha_d) \in \mathbb{R}^d$. Then we write

$$g(x) = b + x^{(1)}w_1 + x^{(2)}w_2 + \ldots + x^{(d)}w_d = \langle w, x \rangle + b.$$

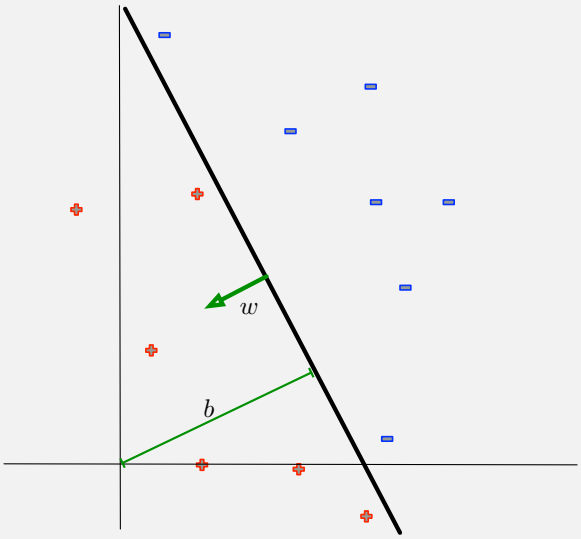
We can now interpret (w, b) as defining a halfspace in $\mathbb{R}^d$. Here w is the normal of that halfspace boundary (the single direction orthogonal to it) and b is the distance from the origin $\mathbf{0} = (0, 0, \ldots, 0)$ to the halfspace boundary in the direction $w/\|w\|$. Because w is normal to the halfspace boundary, b is also distance from the closest point on the halfspace boundary to the origin (in any direction).

We typically ultimately use w as a unit vector, but it is not important since this can be adjusted by changing b. Let w, b be the desired halfspace with $\|w\| = 1$. Now assume we have another w', b' with $\|w'\| = \beta \neq 1$ and $w = w'/\|w'\|$, so they point in the same direction, and b' set so that they define the same halfspace. This implies $b' = b/\beta$. So the normalization of w can simply be done post-hoc without changing any structure.

Recall, our goal is $g(x) \geq 0$ if $y = +1$ and $g(x) \leq 0$ if $y = -1$. So if x lies directly on the halfspace then $g(x) = 0$.

Example: Linear Separator in $\mathbb{R}^2$

Here we show a set $X \in \mathbb{R}^2$ of 13 points with 6 labeled $+$ and 7 labeled $-$. A linear classifier perfectly separates them. It is defined with a normal direction w (pointing towards the positive points) and an offset b.



Using techniques we have already learned, we can immediately apply two approaches towards this problem.

Linear classification via linear regression. For each data points $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, we can immediately represent x_i as the value of d explanatory variables, and y_i as the single dependent variable. Then we can set up a $n \times (d+1)$ matrix $\tilde{X}$, where the ith row is $(1, x_i)$; that is the first coordinate is 1, and the next d coordinates come from the vector x_i. Then with a $y \in \mathbb{R}^n$ vector, we can solve for

$$\alpha = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T y$$

we have a set of $d+1$ coefficients $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_d)$ describing a linear function $g : \mathbb{R}^d \to \mathbb{R}$ defined

$$g(x) = \langle \alpha, (1, x) \rangle.$$

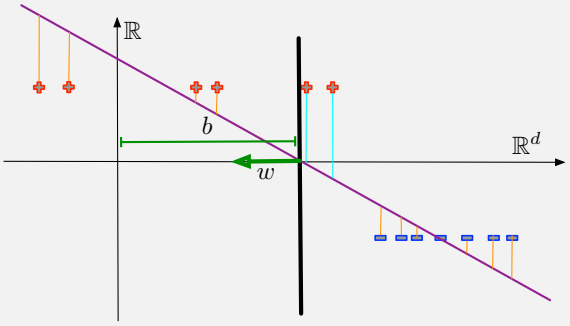
Hence $b = \alpha_0$ and $w = (\alpha_1, \alpha_2, \ldots, \alpha_d)$. For x such that $g(x) > 0$, we predict $y = +1$ and for $g(x) < 0$, we predict $y = -1$.

However, this approach is optimizing the wrong problem. It is minimizing how close our predictions $g(x)$ is to -1 or $+1$, by minimizing the sum of squared errors. But our goal is to minimize the number of mispredicted values, not the numerical value.

 copyright: Jeff M. Phillips

Example: Linear Regression for Classification

We show 6 positive points and 7 negative points in $\mathbb{R}^d$ mapped to $\mathbb{R}^{d+1}$. All of the d-coordinates are mapped to the x-axis. The last coordinate is mapped to the y-axis and is either $+1$ (a positive points) or -1 (a negative points). Then the best linear regression fit is shown, and the points where it has y-coordinate 0 defines the boundary of the halfspace. Note, despite there being a linear separator, this method misclassifies two points because it is optimizing the wrong measure.



Linear classification via gradient descent. Since, the linear regression SSE cost function is not the correct one, what is the correct one? We might define a cost function Δ

$$\Delta(g,(X,y)) = \sum_{i=1}^{n}(1 - \mathbb{1}(\mathsf{sign}(y_i) = \mathsf{sign}(g(x_i)))$$

which uses the *identity function* $\mathbb{1}$ (defined $\mathbb{1}(\textsc{True}) = 1$ and $\mathbb{1}(\textsc{False}) = 0$) to represent the *number* of misclassified points. This is what we would like to minimize.

Unfortunately, this function is discrete, so it does not have a useful (or well-defined) gradient. And, it is also not convex. Thus, encoding g as a $(d+1)$-dimensional parameter vector $(b,w) = \alpha$ and running gradient descent is not feasible.

However, most classification algorithms run some variant of gradient descent. To do so we will use a different cost function as a proxy for Δ, called a *loss function*. We explain this next.

9.1.1 Loss Functions

To use gradient descent for classifier learning, we will use a proxy for Δ called a *loss* functions $\mathcal{L}$. These are sometimes implied to be convex, and their goal is to approximate Δ. And in most cases, they are *decomposable*, so we can write

$$\begin{aligned}\mathcal{L}(g,(X,y)) &= \sum_{i=1}^{n} \ell_i(g,(x_i,y_i)) \\ &= \sum_{i=1}^{n} \ell_i(z_i) \text{ where } z_i = y_i g(x_i).\end{aligned}$$

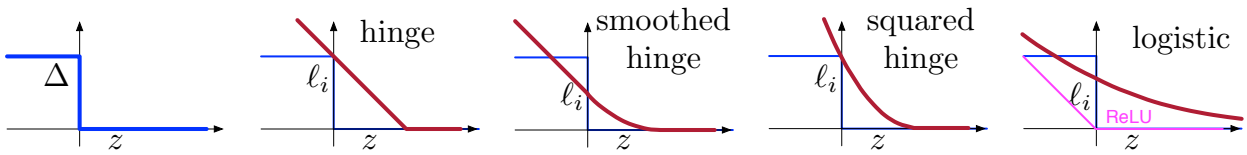
Note that the clever expression $z_i = y_i g(x_i)$ handles when the function $g(x_i)$ correctly predicts the positive or negative example in the same way. If $y_i = +1$, and correctly $g(x_i) > 0$, then $z_i > 0$. On the other hand, if $y_i = -1$, and correctly $g(x_i) < 0$, then also $z_i > 0$. For instance, the desired cost function, Δ is written

$$\Delta(z) = \begin{cases} 0 & \text{if } z \geq 0 \\ 1 & \text{if } z < 0. \end{cases}$$

 copyright: Jeff M. Phillips

Most *loss functions* $\ell_i(z)$ which are convex proxies for Δ mainly focus on how to deal with the case $z_i < 0$ (or $z_i < 1$). The most common ones include:

- hinge loss: $\ell_i = \max(0, 1 - z)$
- smoothed hinge loss: $\ell_i = \begin{cases} 0 & \text{if } z \geq 1 \\ (1-z)^2/2 & \text{if } 0 < z < 1 \\ \frac{1}{2} - z & \text{if } z \leq 0 \end{cases}$
- squared hinge loss: $\ell_i = \max(0, 1 - z)^2$
- logistic loss: $\ell_i = \ln(1 + \exp(-z))$



The hinge loss is the closest convex function to Δ; in fact it strictly upper bounds Δ. However, it is non-differentiable at the "hinge-point," (at $z = 1$) so it takes some care to use it in gradient descent. The smoothed hinge loss and squared hinge loss are approximations to this which are differentiable everywhere. The squared hinge loss is quite sensitive to outliers (similar to SSE). The smoothed hinge loss (related to the Huber loss) is a nice combination of these two loss functions.

The logistic loss can be seen as a continuous approximation to the ReLU (rectified linear unit) loss function, which is the hinge loss shifted to have the hinge point at $z = 0$. The logistic loss also has easy-to-take derivatives (does not require case analysis) and is smooth everywhere. Minimizing this loss for classification is called *logistic regression*.

9.1.2 Cross Validation and Regularization

Ultimately, in running gradient descent for classification, one typically defines the overall cost function f also using a regularization term $r(\alpha)$. For instance $r(\alpha) = \|\alpha\|^2$ is easy to use (has nice derivatives) and $r(\alpha) = \|\alpha\|_1$ (the L_1 norm) induces sparsity (for reasons not covered in this text). In general, the regularizer typically penalizes larger values of α, resulting in some bias, but less over-fitting of the data.

The regularizer $r(\alpha)$ is combined with a loss function $\mathcal{L}(g_\alpha, (X, y)) = \sum_{i=1}^n \ell_i(g_\alpha, (x_i, y_i))$ as

$$f(\alpha) = \mathcal{L}(g_\alpha, (X, y)) + \eta r(\alpha),$$

where $\eta \in \mathbb{R}$ is a *regularization parameter* that controls how drastically to regularize the solution.

Note that this function $f(\alpha)$ is still decomposable, so one can use batch, incremental, or most commonly stochastic gradient descent.

Cross-validation. Backing up a bit, the *true* goal is not minimizing f or $\mathcal{L}$, but predicting the class for new data points. For this, we again assume all data is drawn iid from some fixed but unknown distribution. To evaluate how well out results generalizes, we can use cross-validation (holding out some data from the training, and calculating the expected error of Δ on these held out "testing" data points).

We can also choose the regularization parameter η by choosing the one that results in the best generalization on the test data after training using each on some training data.

 copyright: Jeff M. Phillips

9.2 Perceptron Algorithm

Of the above algorithms, generic linear regression is not solving the correct problem, and gradient descent methods do not really use any structure of the problem. In fact, as we will see we can replace the linear function $g_\alpha(x) = \langle \alpha, (1, x) \rangle$ with any function g (even non-linear ones) as long as we can take the gradient.

Now we will introduce the *perceptron algorithm* which explicitly uses the linear structure of the problem. (Technically, it only uses the fact that there is an inner product – which we will exploit in generalizations.)

Simplifications: For simplicity, we will make several assumptions about the data. First we will assume that the best linear classifier (w^*, b^*) defines a halfspace whose boundary passes through the origin. This implies $b^* = 0$, and we can ignore it. This is basically equivalent to (for data point $(x_i', y_i) \in \mathbb{R}^{d'} \times \mathbb{R}$ using $x_i = (1, x_i') \in \mathbb{R}^d$ where $d' + 1 = d$.

Second, we assume that for all data points (x_i, y_i) that $\|x_i\| \leq 1$ (e.g., all data points live in a unit ball). This can be done by choosing the point $x_{\max} \in X$ with largest norm $\|x_{\max}\|$, and dividing all data points by $\|x_{\max}\|$ so that point has norm 1, and all other points have smaller norms.

Finally, we assume that there exists a perfect linear classifier. One that classifies each data point to the correct class. There are variants to deal with the cases without perfect classifiers, but are beyond the scope of this text.

The algorithm. Now to run the algorithm, we start with some normal direction w (initialized as any positive point), and then add mis-classified points to w one at a time.

Algorithm 9.2.1 Perceptron(X, y)

Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$
repeat
  For any (x_i, y_i) such that $y_i \langle x_i, w \rangle < 0$ (is mis-classified) : update $w \leftarrow w + y_i x_i$
until (no mis-classified points **or** T steps)
return $w \leftarrow w / \|w\|$

Basically, if we find a mis-classified point (x_i, y_i) and $y_i = +1$, then we set $w = w + x_i$. This "pushes" w more in the direction of x_i, but also makes it longer. Having w more in the direction of w, tends to make it have dot-product (with a normalized version of w) closer to 1.

Similar, if we find a mis-classified point (x_i, y_i) with $y_i = -1$, the we set $w = w - x_i$; this pushes the negative of w more towards x_i, and thus w more away from x_i, and thus its dot product more likely to be negative.

 copyright: Jeff M. Phillips

Implementation Hints:
To implement the Perceptron algorithm, inside the inner loop we need to find some misclassified point (x_i, y_i), if one exists. This can require another implicit loop. A common approach would be to, for some ordering of points $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ keep an iterator index i that is maintained outside the **repeat-until** loop. It is modularly incremented every step: it loops around to $i = 1$ after $i = n$. That is, the algorithm keeps cycling through the data set, and updating w for each misclassified point if observes.

Algorithm: Perceptron(X, y)

Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$; Set $i = 1; t = 0;$ LAST-UPDATE $= 1$
repeat
  if $(y_i \langle x_i, w \rangle < 0)$
    $w \leftarrow w + y_i x_i$
    $t = t + 1;$ LAST-UPDATE $= i$
  $i = i + 1 \mod n$
until $(t = T$ or LAST-UPDATE $= i)$
return $w \leftarrow w/\|w\|$

The margin. To understand why the perceptron works, we need to introduce the concept of a margin. Given a classifier (w, b), the *margin* is

$$\gamma = \min_{(x_i, y_i) \in X} y_i(\langle w, x_i \rangle + b).$$

Its the minimum distance of any data point x_i to the boundary of the halfspace. In this sense the optimal classifier (or the maximum margin classifier) (w^*, b^*) is the one that maximizes the margin

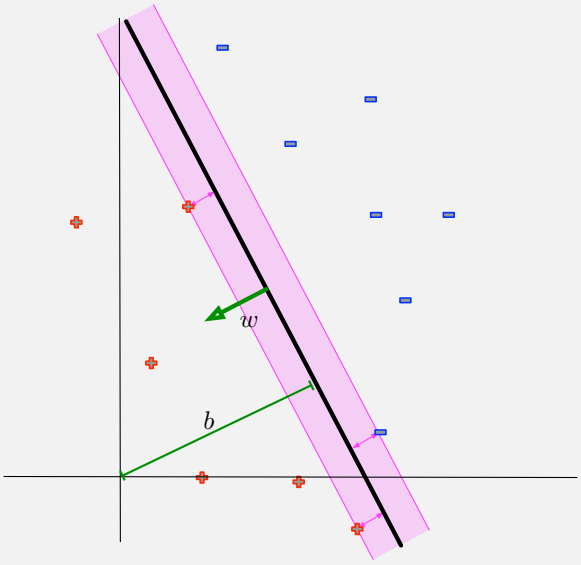
$$(w^*, b^*) = \arg\max_{(w,b)} \min_{(x_i, y_i) \in X} y_i(\langle w, x_i \rangle + b)$$
$$\gamma^* = \min_{(x_i, y_i) \in X} y_i(\langle w^*, x_i \rangle + b^*).$$

A max-margin classifier, is one that not just classifies all points correctly, but does so with the most "margin for error." That is, if we perturbed any data point or the classifier itself, this is the classifier which can account for the most perturbation and still predict all points correctly. It also tends to *generalize* (in the cross-validation sense) to new data better than other perfect classifiers.

 copyright: Jeff M. Phillips

Example: Margin of Linear Classifier

For a set X of 13 points in $\mathbb{R}^2$, and a linear classifier defined with (w, b). We illustrate the margin in pink. The margin $\gamma = \min_{(x_i,y_i)} y_i(\langle w, x_i\rangle + b)$. The margin is drawn with an $\leftrightarrow$ for each support point.



The maximum margin classifier (w^*, b^*) for $X \subset \mathbb{R}^d$ can always be defined uniquely by $d + 1$ points (at least one negative, and at least one positive). These points $S \subset X$ are such that for all $(x_i, y_i) \in S$

$$\gamma^* = y_i\langle w^*, x_i\rangle + b.$$

These are known as the *support points*, since they "support" the margin strip around the classifier boundary.

 copyright: Jeff M. Phillips

Geometry of Why Perceptron Works

Here we will show that after at most $T = (1/\gamma^*)^2$ steps (where γ^* is the margin of the maximum margin classifier), then there can be no more mis-classified points.

To show this we will bound two terms as a function of t, the number of mistakes found. The terms are $\langle w, w^* \rangle$ and $\|w\|^2 = \langle w, w \rangle$; this is before we ultimately normalize w in the **return** step.
First we can argue that $\|w\|^2 \leq t$, since each step increases $\|w\|^2$ by at most 1:

$$\langle w+y_i x_i, w+y_i x_i \rangle = \langle w, w \rangle + (y_i)^2 \langle x_i, x_i \rangle + 2 y_i \langle w, x_i \rangle \leq \langle w, w \rangle + 1 + 0.$$

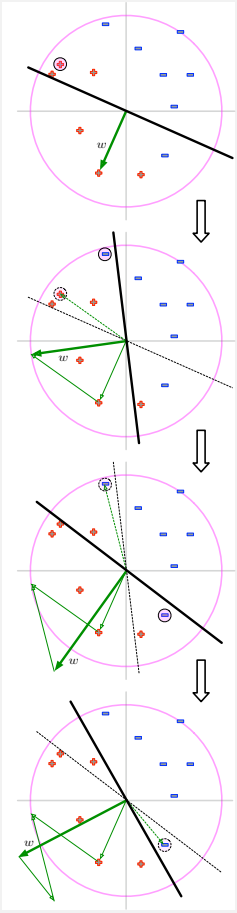
This is true since each $\|x_i\| \leq 1$, and if x_i is mis-classified, then $y_i \langle w, x_i \rangle$ is negative.
Second, we can argue that $\langle w, w^* \rangle \geq t\gamma^*$ since each step increases it by at least γ^*. Recall that $\|w^*\| = 1$

$$\langle w + y_i x_i, w^* \rangle = \langle w, w^* \rangle + (y_i) \langle x_i, w^* \rangle \geq \langle w, w^* \rangle + \gamma^*.$$

The inequality follows from the margin of each point being at least γ^* with respect to the max-margin classifier w^*.
Combining these facts ($\langle w, w^* \rangle \geq t\gamma^*$ and $\|w\|^2 \leq t$) together we obtain

$$t\gamma^* \leq \langle w, w^* \rangle \leq \langle w, \frac{w}{\|w\|} \rangle = \|w\| \leq \sqrt{t}.$$

Solving for t yields $t \leq (1/\gamma^*)^2$ as desired.



9.3 Kernels

It turns out all we need to get any of the above perceptron machinery to work is a well-defined (generalized) inner-product. For two vectors $p = (p_1, \ldots, p_d), x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, we have always used as the inner product:

$$\langle p, x \rangle = \sum_{i=1}^{d} p_i \cdot x_i.$$

However, we can define inner products more generally as a kernel $K(p, x)$. For instance, we can use the following non-linear functions

- $K(p, x) = \exp(-\|p - x\|^2/\sigma^2)$ for the Gaussian kernel, with bandwidth σ,
- $K(p, x) = \exp(-\|p - x\|/\sigma)$ for the Laplace kernel, with bandwidth σ, and
- $K(p, x) = (\langle p, x \rangle + c)^r$ for the polynomial kernel of power r, with control parameter $c > 0$.

Are these linear classifiers? No. In fact, this is how you model various forms of non-linear classifiers. The "decision boundary" is no longer described by the boundary of a halfspace. For the polynomial kernel, the boundary must now be a polynomial surface of degree r. For the Gaussian and Laplace kernel it can be even more complex; the σ parameter essentially bounds the curvature of the boundary.

copyright: Jeff M. Phillips

9.3.1 The Dual: Mistake Counter

To use a more general kernel within the Perceptron algorithm, we need a different interpretation of how to keep track of the weight vector w. Recall, that each step we increment w by $y_i x_i$ for some misclassified data point (x_i, y_i). Instead we will maintain a length n vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, where α_i represents the number of times that data point (x_i, y_i) has been misclassified. That we can rewrite

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i.$$

That is, instead of directly maintaining w, we maintain a length n set of counters α, and keep track of which signed data points $y_i x_i$ to be added to reconstruct w as needed. Now in the linear case the function g, applied to a new data point $p \in \mathbb{R}^d$ becomes

$$g(p) = \langle w, p \rangle = \langle \sum_{i=1}^{n} \alpha_i y_i x_i, p \rangle = \sum_{i=1}^{n} \alpha_i y_i \langle x_i, p \rangle.$$

This seems wasteful to keep such a large vector $\alpha \in \mathbb{R}^n$ around, especially if the number of data points n becomes very large. In contrast, the size of the original version $w \in \mathbb{R}^d$ does not change as the dataset increases. However, we only need to keep track of the non-zero elements of α, and if we run perceptron, there are at most $(1/\gamma^*)^2$ of these. So this is not significantly more space, depending on the relationship between d and $1/\gamma^*$.

The beauty of this form, is that now we can easily replace $\langle x_i, p \rangle$ with any other kernel $K(x_i, p)$. That is, the function $g(p)$ now becomes generalized to

$$g(p) = \langle w, p \rangle = \sum_{i=1}^{n} \alpha_i y_i K(x_i, p).$$

Note this g is precisely a kernel density estimate, with some elements having negative weights (if $y_i = -1$). Then a point p is classified as positive if $g(p) > 0$ and negative otherwise.

9.3.2 Feature Expansion

If the margin is small, the data is not separable, or we simply do not want to deal with the unknown size of a mistake counter, there is another option to use these non-linear kernels. We can take all data, and apply a non-linear transformation to a higher-dimensional space so the problem is linear again.

For a polynomial kernel, on d-dimensional data points, this is equivalent to the polynomial regression expansion, described in Chapter **5.3**. For a two dimensional data point $p = (p_1, p_2)$, we map this to a 5-dimensional space

$$p \mapsto q = (q_1 = p_1, q_2 = p_2, q_3 = p_1^2, q_4 = p_1 p_2, q_5 = p_2^2).$$

Then we search over a 6-dimensional parameter space (b, w) with $w = (w_1, w_2, w_3, w_4, w_5)$ and (with an abuse of notation, since the dimension of w is not d) the kernel defined $K(p, w) = \langle q, w \rangle$. More specifically, the z associated with a data point (p, y) as input to a loss function $\ell(z)$ is defined

$$z = y \cdot (K(p, w) + b) = y \cdot (\langle q, w \rangle + b).$$

Note that the number of free parameters in the feature expansion version of the polynomial kernel is larger than when retaining the kernel in the form $K(x, p) = (\langle x, p \rangle + b)^r$, when it is only $d + 1$. In particular,

when $p \in \mathbb{R}^d$ and the polynomial degree r is large, then this dimensionality can get high very quickly; the dimension of q is $O(q^r)$.

Such expansion is also possible for many other radius basis kernel (e.g., Gaussian, Laplace), but it is only approximate and randomized. Usually it requires the dimensionality of q to be about 100 or more to get a faithful representation.

Like in the case of regression, overfitting can be an issue. However, this effect can be mostly controlled by the regularization term. In the case of the polynomial kernel limiting the polynomial degree r, prevents too complex a model. With the Gaussian and Laplace, and other similar kernels, the increased dimensionality does not lead to overfitting, but too small a value of σ may. In both cases these parameters (r and σ) can be appropriately chosen with cross-validation.

9.3.3 Support Vector Machines

A more general way to work with complex kernels is called a *support vector machine* or SVM. Like with the illustration of the margin for linear classifiers, there are a small number of data points which determine the margin, the support vectors. Just these points are enough to determine the optimal margin.

In the case of complex non-linear kernels (e.g., Gaussians), all of the points may be support vectors. Worse, the associated linear expanded space, the result of complete variable expansion, is actually infinite! This means, the true weight vector w would be infinite as well, so there is no feasible way to run gradient descent on its parameters.

However, in most cases the actual number of support vectors is small. Thus it will be useful to *represent* the weight vector w as a linear combination of these support vectors, without every explicitly constructing them. Consider a linear expansion of a kernel K to an m-dimensional space (think of m as being sufficiently large that it might as well be infinite). However, consider if there are only k support vectors $\{s_1, s_2, \ldots, s_k\} \subset X$ where X is the full data set. Each support vector $s_i \in \mathbb{R}^d$ has a representation $q_i \in \mathbb{R}^m$. But the normal vector $w \in \mathbb{R}^m$ can be written as a linear combination of the q_is; that is, for some parameters $\alpha_1, \alpha_2, \ldots$, we must be able to write

$$w = \sum_{i=1}^{k} \alpha_i q_i$$

Thus, given the support vectors $\{s_1, \ldots, s_k\}$ we can represent w in the span of S (and the origin), reparametrized as a k-dimensional vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$. This α vector is precisely the mistake counter for only the support vectors (the non-zero components), although in this case the coordinates need not be integers.

More concretely, we can apply this machinery without ever constructing the q_i vectors. Each can be implicitly represented as the function $q_i = K(s_i, \cdot)$. Recall, we only ever need to use the q_i in $\langle q_i, w \rangle$. And we can expand w to

$$w = \sum_{i=1}^{k} \alpha_i q_i = \sum_{i=1}^{k} \alpha_i K(s_i, \cdot).$$

Given this expansion, if we consider a special class of kernels called "reproducing kernels" which include Gaussian and Laplace, then we have a super-cool property:

$$K(w, p) = \sum_{i=1}^{k} \alpha_i K(s_i, p).$$

Ultimately, for a data point (p, y), the z in the loss function $\ell(z)$ is defined

$$z = yK(w, p) = y \sum_{i=1}^{k} \alpha_i K(s_i, p).$$

There are multiple ways to actually optimize SVMs: the task of finding the support vectors $S = \{s_1, \ldots, s_k\}$, and assigning their weights $\alpha = \{\alpha_1, \ldots, \alpha_k\}$. One is to run the kernelized Perceptron algorithm, as outlined above. Alternatively, given a fixed set of support vectors S, one can directly optimize over α using gradient descent, including any loss function and regularizer as before with linear classifiers. Thus, if we do not know S, we can just assume $S = X$, the full data set. Then we can apply standard gradient descent over $\alpha \in \mathbb{R}^n$. As mentioned, in most cases, most α_i values are 0 (and those close enough to 0 can often be rounded to 0). Only the points with non-zero weights are kept as support vectors.

Alternatively, stochastic gradient descent works like perceptron, and may only use a fraction of the data points. If we use a version of Hinge Loss, only misclassified points or those near the boundary have a non-zero gradient. The very strongly classified points have zero gradient, and the associated α_i coordinates may remain 0. The proper choice of loss function, and regularizer, can induces sparsity on the α values; and the data points not used are not support vectors.

9.4 kNN Classifiers

Now for something completely different. There are many ways to define a classifier, and we have just touched on some of them. These include decision trees (which basically just ask a series of yes/no questions and are very interpretable) to deep neural networks (which are more complex, far less interpretable, but can achieve more accuracy). We will describe one more simple classifier.

The k-NN classifier (or *k-nearest neighbors classifier*) works as follows. Choose a scalar parameter k (it will be far simpler to choose k as an odd number, say $k = 5$). Next define a *majority* function $\mathsf{maj} : \{-1, +1\}^k \to \{-1, +1\}$. For a set $Y = (y_1, y_2, \ldots, y_k) \in \{-1, +1\}^k$ it is defined

$$\mathsf{maj}(Y) = \begin{cases} +1 & \text{if more than } k/2 \text{ elements of } Y \text{ are } +1 \\ -1 & \text{if more than } k/2 \text{ elements of } Y \text{ are } -1. \end{cases}$$

Then for a data set X where each element $x_i \in X$ has an associated label $y_i \in \{-1, +1\}$, define a k-nearest neighbor function $\phi_{X,k}(q)$ that returns the k points in X which are closest to a query point q. Next let sign report y_i for any input point x_i; for a set of inputs x_i, it returns the set of values y_i.

Finally, the k-NN classifier is

$$g(q) = \mathsf{maj}(\mathsf{sign}(\phi_{X,k}(q))).$$

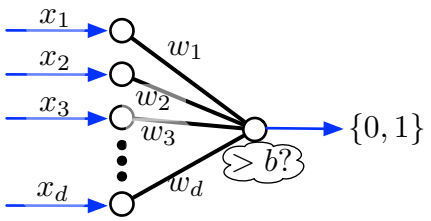
That is, it finds the k-nearest neighbors of query point q, and considers all of the class labels of those points, and returns the majority vote of those labels.

A query point q near many other positive points will almost surely return $+1$, and symmetrically for negative points. This classifier works surprisingly well for many problems but relies on a good choice of distance function to define $\phi_{X,k}$.

Unfortunately, the model for the classifier depends on all of X. So it may take a long time to evaluate on a large data set X. In contrast the functions g for non-kernelized methods above take $O(d)$ time to evaluate for points in $\mathbb{R}^d$, and thus are very efficient.

9.5 Neural Networks

A *neural network* is a learning algorithm intuitively based on how a neuron works in the brain. A neuron takes in a set of inputs $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, weights each input by a corresponding scalar $w = (w_1, w_2, \ldots, w_d)$ and "fires" a signal if the total weight $\sum_{i=1}^{d} w_i x_i$ is greater than some threshold b.



$$\sum_{j=1}^{d} w_i \cdot x_i - b = \langle x, w \rangle - b > 0?$$

A neural network, is then just a network or graph of neurons like these. Typically, these are arranged in layers. In the first layer, there may be d input values $x_1, x_2, \ldots, x_d$. These may provide the input to t neurons (each neuron might use fewer than all inputs). Each neuron produces an output $y_1, y_2, \ldots, y_t$. These outputs then serve as the input to the second layer, and so on.

In a neural net, typically each x_i and y_i is restricted to a range $[-1, 1]$ or $[0, 1]$ or $[0, \infty)$, not just the two classes $\{-1, +1\}$. Since a linear function does not guarantee this of its output, instead of a binary threshold, to achieve this at the output of each node, they typically add an *activation function* $\phi(y)$. Common ones are

- hyperbolic tangent : $\phi(y) = \tanh(y) = \frac{e^y - e^{-y}}{e^y + e^{-y}} \in [-1, 1]$
- sigmoid : $\phi(y) = \frac{1}{1+e^{-y}} = \frac{e^y}{e^y+1} \in [0, 1]$
- ReLu : $\phi(y) = \max(0, y) \in [0, \infty)$.

These functions are not linear, and nor binary. They act as a "soft" version of binary. The hyperbolic tangent and sigmoid stretch values near 0 away from 0. Large values stay large (in the context of the range). So it makes most values almost on the boundaries of the range. And importantly, they are differentiable.

The ReLu has become very popular. It is not everywhere differentiable, but is convex. Its basically the most benign version of activation, and is more likely the original neuron, in that if the value y is negative, it gets snapped to 0. If its positive, it keeps its value.

A two-layer neural network is already a powerful way to build classifiers, and with enough neurons in the middle layer, is capable of learning *any function*. However, *deep neural nets* with 3 or often many more layers (say 20 or 100 or more) have become very popular due to their effectiveness in many tasks, ranging from image classification to language understanding. To be effective, typically, this requires heavy engineering in how the layers are defined, and how the connections between layers are made.

Once the connections are determined, then the goal is to learn the weights on each neuron so that for a given input, a final neuron fires if the input satisfies some pattern (e.g., the input are pixels to a picture, and it fires if the picture contains a car). This is theorized to be "loosely" how the human brain works. Although, neural nets have pretty much diverged in how they learn from attempts to replicate the structure and process of learning in the human brain.

Training. Given a data set X with labeled data points $(x, y) \in X$ (with $x \in \mathbb{R}^d$ and $y \in \{-1, +1\}$), we already know how to train a single neuron so for input x it tends to fire if $y = 1$ and not fire if $y = -1$. It is just a linear classifier! So, we can use the perceptron algorithm, or gradient descent with a well-chosen loss function.

However, for neural networks to attain more power than simple linear classifiers, they need to be at least two layers, and are often deep (e.g., for "deep learning") with 20, 100, or more layers. For these networks, the perceptron algorithm no longer works since it does not properly propagate across layers. However, a version of gradient descent called *back-propagation* can be used. In short, it computes the gradient across the edge weights in a network by chaining partial derivatives backwards through the network.

Training deep nets to work can be quite finicky. Their optimization function is not convex, and without various training tricks, it can be very difficult to find a good global set of weights. Indeed the best and "right" methods are an active research area.

 copyright: Jeff M. Phillips

Exercises

Q9.1: Consider the following "loss" function. $\ell_i(z_i) = (1 - z_i)^2/2$, where for a data point (x_i, y_i) and prediction function f, then $z_i = y_i \cdot f(x_i)$. Predict how this might work within a gradient descent algorithm for classification.

Q9.2: Consider a data set (X, y), where each data point $(x_{1,i}, x_{2,i}, y_i)$ is in $\mathbb{R}^2 \times \{-1, +1\}$. Provide the psuedo-code for the Perceptron Algorithm using a polynomial kernel of degree 2. You can have a generic stopping condition, where the algorithm simply runs for T steps for some parameter T. *(There are several correct ways to do this, but be sure to explain how to use a polynomial kernel clearly.)*

Q9.3: Consider a set of 1-dimensional data points

$$(x_1 = 0, y_1 = +1)\ (x_2 = 1, y_1 = -1)\ (x_3 = 2, y_1 = +1)\ (x_4 = 4, y_1 = +1)$$

$$(x_5 = 6, y_1 = -1)\ (x_6 = 7, y_1 = -1)\ (x_7 = 8, y_1 = +1)\ (x_8 = 9, y_1 = -1)$$

Predict **-1** or **+1** using a kNN (k-nearest neighbor) classifier with $k = 3$ on the following queries.

1. $x = 3$
2. $x = 9$
3. $x = -1$

Q9.4: Consider the following Algorithm 1, called the *Double-Perceptron*. We will run this on an input set X consisting of points $X \in \mathbb{R}^{n \times d}$ and corresponding labels $y \in \{-1, +1\}$.

Algorithm 9.5.1 Double-Perceptron(X)

Initialize $w = y_i x_i$ for any $(x_i, y_i) \in (X, y)$
repeat
  For any (x_i, y_i) such that $y_i \langle x_i, w \rangle < 0$ (is mis-classified) : update $w \leftarrow w + 2 \cdot y_i x_i$
until (no mis-classified points **or** T steps)
return $w \leftarrow w/\|w\|$

For each of the following questions, the answer can be **faster**, **slower**, **the same**, or **not at all**. And should be accompanied with an explanation.

1. Compared with Algorithm 9.2.1 (Perceptron) in the notes, explain how this algorithm with converge.
2. Next consider transforming the input data set X (not the y labels) so that all coordinates are divided by 2. Now if we run *Double-Perceptron* how will the results compare to regular Perceptron (Algorithm 9.2.1) on the original data set X.
3. Finally, consider taking the original data set (X, y) and multiplying all entries in y by -1, then running the original Perceptron algorithm. How will the convergence compare to running the same Perceptron algorithm, on the original data set.

Q9.5: Consider a matrix $A \in \mathbb{R}^{n \times 4}$. Each row represents a customer (there are n customers in the database). The first column is the age of the customer in years, the second column is the number of days since the customer entered the database, the third column is the total cost of all purchases ever by the customer

 copyright: Jeff M. Phillips

in dollars, and the last column is the total profit in dollars generated by the customer. So each column has a different unit.

For each of the following operations, decide if it is **reasonable** or **unreasonable**.

1. Run simple linear regression using the first three columns to build a model to predict the fourth column.
2. Use k-means clustering to group the customers into 4 types using Euclidean distance between rows as the distance.
3. Use PCA to find the best 2-dimensional subspace, so we can draw the customers in a $\mathbb{R}^2$ in way that has the least projection error.
4. Use the linear classification to build a model based on the first three columns to predict if the customer will make a profit $+1$ or not -1.

Q9.6: Consider a "loss" function, called an *double-hinged loss function*

$$\ell_i(z) = \begin{cases} 0 & \text{if } z > 1 \\ 1 - z & \text{if } 0 \leq z \leq 1 \\ 1 & \text{if } z \leq 0. \end{cases}$$

where the overall cost for a dataset (X, y), given a linear function $g(x) = \langle (1, x), \alpha \rangle$ is defined $\mathcal{L}(g, (X, y)) = \sum_{i=1}^{n} \ell_i(y_i \cdot g(x_i))$.

1. What problems might a gradient descent algorithm have when attempting to minimize $\mathcal{L}$ by choosing the best α?
2. Explain if the problem would be better or worse using stochastic gradient descent?

copyright: Jeff M. Phillips